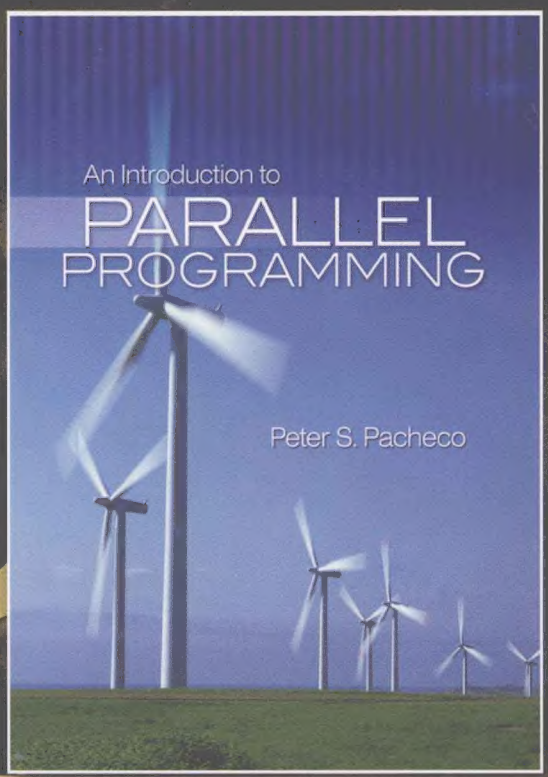
HZ BOOKS 华章教育
ELSEVIER

计 算 机 科 学 丛 书

# 并行程序设计导论

（美）Peter S. Pacheco 著　邓倩妮 等译

## An Introduction to Parallel Programming




机械工业出版社
China Machine Press

# 并行程序设计导论

## An Introduction to Parallel Programming

毫无疑问，随着多核处理器和云计算系统的广泛应用，并行计算不再是计算世界中被束之高阁的偏门领域。并行性已经成为有效利用资源的首要因素，Peter Pacheco撰写的这本新教材对于初学者了解并行计算的艺术和实践很有帮助。

—— Duncan Buell，南卡罗来纳大学计算机科学与工程系

本书阐述了两个越来越重要的领域：使用Pthreads和OpenMP进行共享内存编程，以及使用MPI进行分布式内存编程。更重要的是，通过指出可能出现的性能错误，强调好的编程实现的重要性。这些主题包含在计算机科学、物理学和数学等多个学科中。各章包含了大量不同难易程度的编程习题。对于希望学习并行编程技巧、更新知识面的学生或专业人士来说，本书是一本理想的参考书。

—— Leigh Little，纽约州立大学布罗科波特学院计算机科学系

并行编程已不仅仅是面向专业技术人员的一门学科。如果想要全面开发集群和多核处理器的计算能力，那么学习分布式内存和共享内存的并行编程技术是不可或缺的。本书是一本精心撰写的、全面介绍并行计算的书籍。作者循序渐进地展示了如何利用MPI、Pthreads 和OpenMP开发高效的并行程序，教给那些专业知识有限、没有并行化经验的读者如何开发、调试分布式内存和共享内存的程序，以及对程序进行性能评估。

### 本书特色

- 采用教程形式，从简短的编程实例起步，一步步编写更有挑战性的程序。
- 重点介绍分布式内存和共享内存的程序设计、调试和性能评估。
- 使用MPI、Pthreads 和OpenMP等编程模型，强调实际动手开发并行程序。

### 作者简介

**Peter S. Pacheco** 拥有佛罗里达州立大学数学专业博士学位。曾担任旧金山大学计算机系主任，目前是旧金山大学数学系主任。近20年来，一直为本科生和研究生讲授并行计算课程。


英文版
书号：978-7-111-35838-2
定价：65.00元

本书译自原版An Introduction to Parallel Programming，并由Elsevier授权出版

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

封面设计：锡 林

上架指导：计算机 并行计算 并行编程

ISBN 978-7-111-39284-2

9 787111 392842

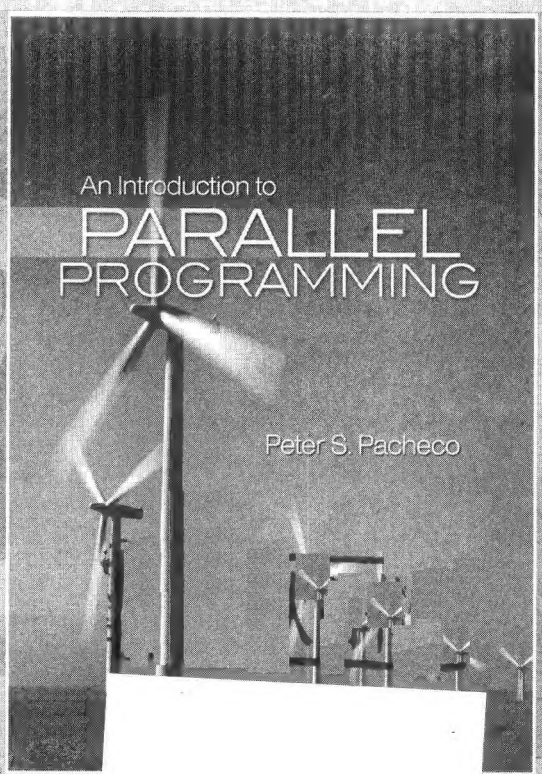
定价：49.00元

机 科 学 丛 书

# 并行程序设计导论

（美）Peter S. Pacheco 著　邓倩妮 等译

## An Introduction to Parallel Programming




机械工业出版社
China Machine Press

本书全面涵盖了并行软件和硬件的方方面面，深入浅出地介绍如何使用 MPI（分布式内存编程）、Pthreads 和 OpenMP（共享内存编程）编写高效的并行程序。各章节包含了难易程度不同的编程习题。

本书可以用做计算机科学专业低年级本科生的专业课程的教材，也可以作为软件开发人员学习并行程序设计的专业参考书。



**本书版权登记号：图字：01-2011-4803**

**图书在版编目（CIP）数据**

并行程序设计导论/（美）帕切克（Pacheco，P. S.）著；邓倩妮等译．—北京：机械工业出版社，2012.8

（计算机科学丛书）

书名原文：An Introduction to Parallel Programming

ISBN 978-7-111-39284-2

Ⅰ. 并…　Ⅱ. ①帕…　②邓…　Ⅲ. 并行程序－程序设计　Ⅳ. TP311.11

中国版本图书馆 CIP 数据核字（2012）第 173261 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：盛思源

北京市荣盛彩色印刷有限公司印刷

2013 年 1 月第 1 版第 1 次印刷

185mm×260mm · 16.5 印张

标准书号：ISBN 978-7-111-39284-2

定价：49.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www. hzbook. com
电子邮件：hzjsj@hzbook. com
联系电话：(010) 88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037


华章教育
华章科技图书出版中心

# 译者序
An Introduction to Parallel Programming

本书在对并行硬件和并行软件进行知识总结之后，着重介绍如何利用 MPI、Pthreads 和 OpenMP 开发高效的并行程序。本书的特点在于：

1）文字流畅，易于理解。本书内容通俗易懂，简洁实用。清晰的概念解释，配以丰富的实例和易懂的代码，对帮助初学者理解并行程序设计的基本手段非常重要，可以帮助读者很快掌握设计并行程序的基本方法。

2）循序渐进，由浅及深。本书分别介绍了如何利用 MPI、Pthreads 和 OpenMP 进行并行程序设计。每一章都从最基本的实例开始示范，再介绍一些常见问题不同的实现方法，最后分析和比较不同实现方法的性能。这不仅能帮助初学者快速掌握并行编程方法，还能让读者进一步学习开发高效并行程序设计的方法。

3）重实践，重开发。各章包含了详细介绍的编程实例，以及不同难易程度的编程习题。

本书不仅适合作为计算机专业并行程序设计的课程教材，对需要通过并行程序设计提高计算性能的其他学科（如物理、机械、生物医药等专业）的技术人员，也可以作为参考手册。

本书由上海交通大学邓倩妮副教授主持翻译定稿。此外，冯叶、曾卫、黄鑫、黄叶伟、戴云晶、王强和吕品也参加了本书部分翻译工作，黄鑫还参与了本书的校对工作，对他们的支持和帮助，在此表示衷心的感谢。

由于时间和水平有限，翻译中难免存在不准确，敬请读者指正。

译者

2012 年 6 月

# 本书赞誉

An Introduction to Parallel Programming

随着多核处理器和云计算系统的广泛应用，毫无疑问，并行计算不再是计算世界中被束之高阁的偏门领域。并行性已经成为有效利用资源的首要因素。由 Peter Pacheco（彼得·帕切克）撰写的这本新教材，对于刚开始学术生涯的学生掌握并行计算的艺术和实践很有帮助。

Duncan Buell

南卡罗来纳州大学计算机科学与工程系

本书阐述了两个越来越重要的领域：使用 Pthreads 和 OpenMP 进行共享内存编程，以及使用 MPI 进行分布式内存编程的基本方法。更重要的是，通过指出可能出现的性能错误，强调好的编程实践的重要性。这些主题包含在计算机科学、物理学和数学等多个学科中。各章包含了大量不同难易程度的编程习题。对希望学习并行编程技巧、更新知识面的学生或专业人士来说，本书是一本理想的参考书。

Leigh Little

纽约州立大学布罗科波特学院计算机科学系

本书是一本精心撰写的、全面介绍并行计算的书籍。学生以及相关领域从业人员会从本书的相关最新信息中获益匪浅。Peter Pacheco 通俗易懂的写作手法，结合各种有趣的实例，使本书引人入胜。在并行计算这个瞬息万变、不断发展的领域里，本书深入浅出、全面地介绍了并行软件和并行硬件的方方面面。

Kathy J. Liszka

阿克伦大学计算机科学系

并行计算就是未来！本书通过实用而有益的例子，介绍了这门复杂的学科。

Andrew N. Sloss，FBCS

ARM 公司顾问工程师，《ARM System Developer's Guide》作者

# 前 言
An Introduction to Parallel Programming

并行硬件已经普及了一段时间。现在已经很难找到一台没有多核处理器的笔记本、台式机或者服务器。在20世纪90年代还是高性能工作站的Beowulf集群，而今已经达到普及程度。与此同时，云计算的出现使得分布式内存系统与台式机一样便于访问。尽管如此，大多数计算机专业的学生在毕业时拥有很少甚至几乎没有任何并行编程经验。虽然许多学院或大学为高年级学生提供并行计算选修课程，但因为计算机科学专业有过多的必修课要求，很多人在毕业时甚至没有写过一个多线程或者多进程的程序。

毫无疑问，这样的现状是需要改变的。虽然许多程序在单核上获得了较满意的性能，但是计算机科学家们必须意识到：并行化有潜力使性能得到巨大提升。当需求提升时，他们应该具备开发这种潜力的能力。

本书旨在部分地解决该问题。它介绍如何使用MPI、Pthreads和OpenMP编写并行程序。MPI、Pthreads和OpenMP是三个广泛应用在并行编程中的应用程序编程接口（Application Programming Interface，API）。本书的预期读者是需要编写并行程序的学生和专业人员。阅读本书仅需要很少的预备知识：大专程度的数学知识和使用C语言编写串行程序的能力。前导知识要求少，因为我们认为学生应该尽快具备编写并行系统的能力。

在旧金山大学，计算机科学专业的学生可以通过学习以本书为教材的课程来达到专业课的要求。“计算机科学导论”是大多数新生在第一个学期学习的课程，本书介绍的课程可以安排为它的后续课程。我们将这门课程作为并行计算的相关课程已经有6年时间。根据我们的经验，学生完全不需要从中、高年级才开始编写并行程序。相反，这门课程十分受欢迎。通过学习这门导论课，学生可以很轻松地将并行性应用于其他课程。

第二学期的新生可以通过课堂学习编写并行程序，而带着目的进行学习的计算机专业人员可以自学并行编程。我们希望本书对于他们是有用的资源。

## 关于本书

正如前面所说的，本书的主要目的是：让那些对计算机科学只有有限背景知识、没有并行性经验的读者学习使用MPI、Pthreads和OpenMP进行并行编程。为了让本书使用起来更灵活，我们尽量让那些对API没有兴趣的读者花费较少的时间就能很容易地阅读剩下的部分。因此，针对这三个API的章节是相互独立的：可以按任意顺序阅读它们，甚至可以跳过一两个章节。但是，这种独立性有一定代价：有些内容会在这些章节中重复提到。当然，重复的内容可以简单浏览或者直接跳过。

没有并行计算经验的读者应该先阅读第1章。第1章尝试用相关的非技术性的语言解释为什么并行系统已经在计算机领域中占有重要地位。这一章还为并行系统和并行编程提供了一个简短的介绍。

第2章介绍计算机硬件和软件的一些技术背景。在API章节开始之前，许多关于硬件的材料可以先粗略浏览。第3章、第4章、第5章分别介绍使用MPI、Pthreads、OpenMP进行编程。

在第6章中，我们开发了两个更长的程序：并行$n$体问题求解和并行树搜索。这两个程序都使用上述的三个API。第7章提供一个简单的列表，给出并行计算各个方面的补充信息。

我们使用C语言来开发程序，因为这三个API都使用C语言接口，同时也因为C语言是相对简单易学的语言，尤其是对于C++和Java程序员来说，他们已经对C的控制结构非常熟悉。

## 课堂使用

本书来源于旧金山大学低年级本科生的课程。这门课满足了计算机科学的专业课要求，同时也是本科生操作系统课程的先修课。本课程唯一的要求是在第一学期的“计算机科学导论”课程中获得 B 及以上成绩，或在第二学期的“计算机科学导论”课程中获得 C 及以上成绩。课程的前四个星期介绍 C 语言编程。由于大部分学生已经编写过 Java 程序，因此课程主要集中在 C 语言中如何使用指针上[⊖]。课程剩下的部分介绍使用 MPI、Pthreads、OpenMP 编程。

上述的大部分内容集中在本书第 1 章、第 3 章、第 4 章、第 5 章中，并在第 2 章和第 6 章中有少量提及。当需求提高时，第 2 章的背景知识应该介绍。比如，在讨论 OpenMP（第 5 章）缓存一致性问题前，应该先介绍第 2 章中缓存的知识。

本课程的作业包括每周作业、5 个编程作业、两次期中考试和一次期末考试。作业中经常涉及编写一个简短的程序或者对现有的程序进行一些小的改进。这些作业的目的是保证学生能够跟上课程，并且根据课堂上的想法得到实际操作经验。这些作业的布置是本课程成功的重要原因。课本中大多数习题与这些简短的作业是相适应的。

编程作业中需要编写更大的程序，但我们一般会为学生提供非常多的指导：我们经常在作业中提供伪代码，并在课堂上讨论较难的部分。这些额外的指导是非常有效的：那些需要花费学生大量时间完成的编程作业变得不再难以提交。期中和期末考试的结果，以及讲授操作系统课程的教师的报告都表明，这门课程对于教授学生如何编写并行程序是非常成功的。

对于其他高级并行计算课程，本书和它的在线辅助材料可以作为补充，许多关于这三个 API 语法和语义的信息都可以作为课外阅读资料。本书也可以作为工程方面的课程或者与计算机科学无关但涉及并行计算的课程的补充材料。

## 辅助材料

关于本书的勘误表和一些相关材料，请访问本书出版社的网站（http：//www. elsevierdirect. com/），那里还可以下载完整的课件、图表、习题答案和编程作业。所有的用户都可以下载课本中讨论过的较长程序。

我们非常感谢读者提出任何发现的错误。如果你发现错误，请发送邮件至 peter@ usfca. edu。

---

⊖ 有趣的是，很多学生认为 C 语言中的指针比 MPI 编程更困难。

# 致 谢

An Introduction to Parallel Programming

在编写本书的过程中，得到了许多人的帮助。非常感谢那些在本书编写初期阅读并提出建议的评阅人：Fikret Ercal (Missouri University of Science and Technology)，Dan Harvey (Southern Oregon University)，Joel Hollingsworth (Elon University)，Jens Mache (Lewis and Clark College)，Don McLaughlin (West Virginia University)，Manish Parashar (Rutgers University)，Charlie Peck (Earlham College)，Stephen C. Renk (North Central College)，Rolfe Josef Sassenfeld (The University of Texas at El Paso)，Joseph Sloan (Wofford College)，Michela Taufer (University of Delaware)，Pearl Wang (George Mason University)，Bob Weems (University of Texas at Arlington)，Cheng-Zhong Xu (Wayne State University)。

我同样非常感谢以下评阅人，他们针对各个章节提出了不同的建议：Duncan Buell (University of South Carolina)，Matthias Gobbert (University of Maryland, Baltimore County)，Krishna Kavi (University of North Texas)，Hong Lin (University of Houston-Downtown)，Kathy Liszka (University of Akron)，Leigh Little (The State University of New York)，Xinlian Liu (Hood College)，Henry Tufo (University of Colorado at Boulder)，Andrew Sloss (Consultant Engineer, ARM)，Gengbin Zheng (University of Illinois)。他们的意见和建议使得本书得到巨大的改进。当然，我个人需要对依然存在的错误和漏洞负责。

Kathy Liszka 为采用本书的教师准备了幻灯片；Jinyoung Choi，我先前的学生，为本书准备了一份答案手册。同样感谢他们。

Morgan Kaufmann 的员工在整个项目中对我提供了非常多的帮助。我尤其感谢编辑 Nate McFadden，他给了我很多宝贵的建议，并做了了不起的工作安排，在过去几年中，他与我讨论了所有遇到的问题，并表现了极大的耐心。同时感谢 Marily Rash 和 Megan Guiney，他们在编写过程中表现得非常迅速有效。

旧金山大学计算机科学和数学系的同事们在编写本书的过程中为我提供了非常多的帮助。这里特别感谢 Gregory Benson 教授：他对于并行计算的理解（尤其是 Pthreads 和信号量）是我宝贵的资源。我也非常感谢我们的系统管理员 Alexey Fedosov 和 Colin Bean，在我编写本书的过程中，他们耐心并有效地处理了出现的所有“紧急情况”。

如果没有我的朋友 Holly Cohn、John Dean、Robert Miller 的鼓励和道义上的支持，我将永远不可能完成本书，他们帮我度过了一段非常困难的时期，我永远感谢他们。

我最大的感谢致予我的学生，是他们向我展示了什么是过于简单，什么是过于困难。总之，他们教会了我如何去教授并行计算。我向他们致以最深切的感谢。

第1章
An Introduction to Parallel Programming

# 为什么要并行计算

从1986年到2002年，微处理器的性能以平均每年50%的速度不断提升［27］。这样史无前例的性能提升，使得用户和软件开发人员只需要等待下一代微处理器的出现，就能够获得应用程序的性能提升。但是，从2002年开始，单处理器的性能提升速度降低到每年大约20%，这个差异是巨大的：如果以每年50%的速度提升，在10年里微处理器的性能会提升60倍，而以20%的速度，10年里只能提升6倍。

此外，性能提升速度的差异也极大地改变了处理器的设计。到2005年，大部分主流的微处理器制造商已决定通过并行处理来快速提升微处理器的性能。他们不再继续开发速度更快的单处理器芯片，而是开始将多个完整的单处理器放到一个集成电路芯片上。

这一变化对软件开发人员带来了重大的影响：大多数**串行**程序是在单个处理器上运行的，不会因为简单地增加更多的处理器就获得极大的性能提高。串行程序不会意识到多个处理器的存在，它们在一个多处理器系统上运行的性能，往往与在多处理器系统的一个处理器上运行的性能相同。

所有这一切引出如下问题：

1）为什么我们要关心并行？单处理器系统不是已经足够快了吗？毕竟每年20%的性能提升也是很可观的。

2）为什么微处理器制造商不能继续研制更快的单处理器系统？为什么要研制**并行系统**？为什么要研制多处理器系统？

3）为什么不能编写程序，将串行程序自动转换成可以充分利用多处理器的**并行程序**？

下面我们简单地回答上述问题。但请记住：有些问题的答案不是一成不变的。例如，每年20%的性能提升对大多数应用程序来说是绰绰有余的了。



## 1.1 为什么需要不断提升的性能

过去几十年中，不断提升的计算能力已经成为许多飞速发展领域（如科学、互联网、娱乐等）的核心力量。例如：人类基因解码、更准确的医疗成像、更快速精确的网络搜索、更真实的电脑游戏，都离不开计算能力的提高。确实，没有早期的提高，现在很多应用的计算能力提升将会很难实现，甚至不可能实现。但是，我们不能满足于现状。随着计算能力的提升，我们要考虑解决的问题也在增加，如下就是一些例子：

- *气候模拟*：为了更好地理解气候变化，我们需要更加精确的计算模型，这种模型必须包括大气、海洋、陆地以及极地冰川之间的相互关系。我们需要对各种因素如何影响全球气候做详细研究。
- *蛋白质折叠*：人们相信错误折叠的蛋白质与亨廷顿病、帕金森病、老年痴呆症等疾病有千丝万缕的联系，但现有的计算性能严重限制了研究复杂分子（如蛋白质）结构的能力。
- *药物发现*：不断提高的计算能力可以从不同方面促进新的医学研究。例如，有许多药物只是对一小部分患者有效。我们可以通过仔细分析疗效欠佳患者的基因来找到替代的药物，但这需要大规模的基因组计算和分析。

- 能源研究：不断提高的计算能力可以为某些技术（如风力涡轮机、太阳能电池和蓄电池）构建更详细的模型。这些模型能够为建立更高效清洁的能源提供信息。
- 数据分析：每天都会产生大量的数据。据估计，全球范围存储的数据每两年翻一番［28］，而这些数据中的大部分在未经分析前是无用的。例如，了解人类 DNA 的核苷酸序列本身是没有用的，而理解这个序列如何影响生长发育，以及它是如何引起疾病的，需要大规模的数据分析。除了基因组学，欧洲核子研究中心（CERN）的大型强子对撞机、医疗成像、天文研究、网络搜索引擎也会产生海量的数据，并需要对这些数据进行大规模分析。

[2] 上述这些问题以及其他问题的解决都需要更强大的计算能力。

## 1.2 为什么需要构建并行系统

单处理器性能大幅度提升的主要原因之一，是日益增加的集成电路晶体管密度（晶体管是电子开关）。随着晶体管尺寸的减小，晶体管的传递速度增快，集成电路整体的速度也增快。但是，随着晶体管速度的增快，它们的能耗也相应增加。大多数能量是以热能的形式消耗，当一块集成电路变得太热的时候，就会变得不可靠。在 21 世纪的第一个 10 年中，用空气冷却的集成电路的散热能力已经达到了极限［26］。

因此，通过继续增快集成电路的速度来提高处理器性能的方法变得不再可行。但是集成电路晶体管的密度还在增加，并且还会持续一段时间。而且，既然计算方法对改善我们现有的方式有潜在的推动作用，那么继续发掘更强的计算能力就是必要而迫切的。最后，如果集成电路制造商不能继续推出更新、更好的产品，那么它很快就会被淘汰。

我们如何利用还在不断增加的晶体管密度？答案是*并行*。集成电路制造商的决策是：与其构建更快、更复杂的单处理器，不如在单个芯片上放置多个相对简单的处理器。这样的集成电路称为**多核**处理器。**核**已经成为中央处理器或者 CPU 的代名词。在这样的设定下，传统的只有一个 CPU 的处理器称为**单核**系统。

## 1.3 为什么需要编写并行程序

大多数为传统单核系统编写的程序无法利用多核处理器。虽然可以在多核系统上运行一个程序的多个实例，但这样意义不大。例如，在多个处理器上运行一个喜爱的游戏程序的多个实例并不是我们需要的。我们需要的是这个程序能够更快地运行，有更加逼真的图像。为了达到这一目的，就需要将串行程序改写为*并行程序*，或者编写一个翻译程序来自动地将串行程序翻译成并行程序，只有这样才能充分利用多核。不幸的是，研究人员在自动将串行程序（例如 C 或C ++ 编写的程序）转换成并行程序上鲜有突破。

这并不令人惊讶。尽管我们可以编写一些程序，让这些程序辨识串行程序的常见结构，并自动将这些结构转换成并行程序的结构，但转化后的并行程序在实际运行时可能很低效。例如，两个 $n \times n$ 的矩阵相乘是由多个点积操作组成的一个序列，但是将矩阵相乘并行化为一系列并行执

[3] 行的点积操作，在很多系统上运行效率并不高。

一个串行程序的高效并行实现可能不是通过发掘其中每一个步骤的高效并行实现来获得，相反，最好的并行化实现可能是通过一步步回溯，然后发现一个全新的算法来获得的。

举例来说，假设我们需要计算 $n$ 个数的值再累加求和，如下是串行代码：

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

现在我们假设有 $p$ 个核，且 $p$ 远小于 $n$，那么每个核能够计算大约 $n/p$ 个数的值并累加求和，以得到部分和：

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value(. . .);
   my_sum += my_x;
}
```

此处的前缀 my_代表每个核都使用自己的私有变量，并且每个核能够独立于其他核来执行该代码块。

每个核都执行完代码后，变量 my_sum 中就会存储调用 Compute_next_value 获得的值的和。例如，假如有 8 个核，$n=24$，24 次调用 Compute_next_value 获得如下的值：

1，4，3　9，2，8　5，1，1　6，2，7　2，5，0　4，1，8　6，5，1　2，3，9

这样存储在 my_sum 中的值将是：

| 核 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| my_sum | 8 | 19 | 7 | 15 | 7 | 13 | 12 | 14 |

这里，我们假定用非负整数 0，1，…，$p-1$ 来标识各个核，$p$ 是核的总数。

当各个核都计算完各自的 my_sum 值后，将自己的结果值发送给一个指定为“master”的核（主核），master 核将收到的部分和累加而得到全局总和：

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
}else{
  send my_x to the master;
}
```

4

在我们的例子中，假如 master 核是 0 号核，它将部分和累加求全局总和 8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95。

但是，可能你已经想到一个更好的方法了——特别是当核的数目比较多的时候，不再由 master 核计算所有部分和的累加工作，可以将各个核两两结对，0 号核将自己的部分和与 1 号核的部分和做加法，2 号核将自己的部分和与 3 号核的部分和做加法，4 号核将自己的部分和与 5 号核的部分和做加法，以次类推。然后，再在偶数核上重复累加部分和：0 号核加上 2 号核，4 号核加上 6 号核，以次类推，如图 1-1 所示。圆圈表示当前核所得到的和，箭头表示一个核将自己的部分和发送给另一个核，加号表示一个核在收到另一个核发送来的部分和后与自己本身的部分和相加。

上述两种计算全局总和的算法中，master 核（0 号核）承担了更多的工作量，整个程序计算全局总和的时间就等于 master 核的计算时间。在 8 个核的情况下，第一种方法中，master 核需要执行 7 次接收操作，而第二种方法中，master 核仅需要执行 3 次接收操作。因此第二种方法比第一种方法快 2 倍，当有更多的核时，两者的差异更大。在 1000 个核的情况下，第一种方法需要 999 次接收和加法操作，而第二种方法只需要 10 次，提高了 100 倍。

5

非常明显，第一种计算全局总和的方法是对串行求和程序的一般化：将求和的工作在核之间平分，等到每个核都计算出部分和之后，master 简单地重复串行程序中基本的串行求和。如果有

$p$ 个核，master 就需要计算 $p$ 个值的总和。而第二种计算全局总和的方法与原来的串行程序没有多大关系。

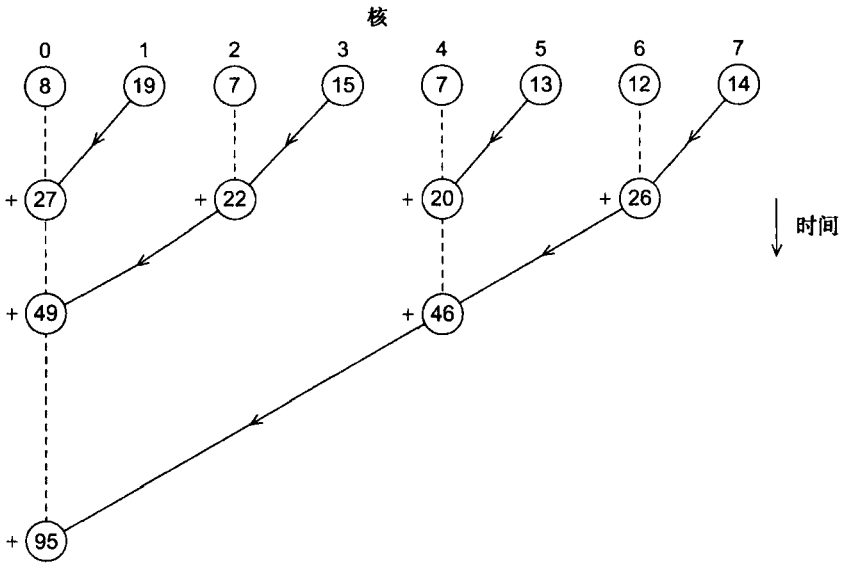



图 1-1　多个核共同计算形成一个全局总和

问题的关键是：除非事先已经定义好这么一个高效的求全局和算法，否则翻译程序不太可能发现第二种计算全局总和的方法。在翻译的时候，通过识别出原来的串行循环，将其替换为预定义好的高效并行求和算法。

我们期望通过编写软件，识别出常见的串行结构，并对其进行有效的**并行化**，使其能够利用多个核。但是，当我们将此原则应用于更复杂的串行程序时，识别结构将变得越来越困难，转换为事先定义好的高效并行化方法也变得越来越不可能。

因此，我们不能再继续简单地编写串行程序，我们必须编写并行程序来发掘多核处理器的潜在性能。

## 1.4　怎样编写并行程序

对于这个问题，有多种可能的解决方案。大部分方案的基本思想都是将要完成的任务分配给各个核。有两种广泛采用的方法：**任务并行**和**数据并行**。任务并行是指将待解决问题所需要执行的各个任务分配到各个核上执行。而数据并行是指将待解决问题所需要处理的数据分配给各个核，每个核在分配到的数据集上执行大致相似的操作。

举例来说，假如 P 教授进行“英国文学调查”的授课，她有 100 个学生，还有 4 个助教（Teaching Assistant，TA），A 先生、B 女士、C 先生、D 女士。学期结束的时候，要进行一次期末测试，这个测试中包括 5 道题。为了给学生打分，P 教授和她的助教可能有如下两种批改方案：每人负责给一个问题打分；或者将学生分成 5 组，每人负责一组，即 20 个学生。

在这两种方案中，P 教授和她的助教充当核的角色。第一种方案可以认为是任务并行的例子。有 5 个任务需要执行，即给第一个问题打分，给第二个问题打分……给第五个问题打分。当然，每个打分人审阅的题目是不一样的，比如：第一个问题是关于莎士比亚的，第二个问题可能是关于米利托的，所以 P 教授和她的助教是在“执行不同的指令”。



第二种方案可以认为是数据并行的例子。“数据”是学生的卷子，在不同的核（打分人）之间平分，每个核执行大致相似的打分“指令”。

我们在 1.3 节中提到过计算全局总和的例子，其中第一部分可以认为是数据并行的一个实例。数据是通过 Compute_next_value 计算得到的值，每个核在所赋予的数据集上执行大致相同的操作：通过调用 Compute_next_value 获取所需的数据，再将数据累加求部分和。计算全局总和例子的第二部分可以认为是任务并行的实例，总共有两个任务：一个任务由 master 核执行，负责接收从其他核传来的部分和，并累加部分和；另一个任务由其他核执行，负责将自己计算得到的部分和传递给 master 核。

当各个核独立工作时，编写并行程序其实与串行程序差不多。但当核之间需要协调工作时，就变得复杂了。在第二个计算全局总和的例子中，尽管图中的树形结构很容易理解，但实际编写代码却比较复杂。详见习题 1.3 和习题 1.4。不幸的是，核之间需要通信的情况是很常见的。

在两个计算全局总和的例子中，协调过程包括**通信**：一个或多个核将自己的部分和结果发送给其他的核。同时，该例子也反映了协调工作应该**负载平衡**，虽然没有给出明确的公式，但很明显，我们希望给每个核分配大致相同数目的数据来计算。如果某个核必须要计算大部分数据，那么其他的核势必比负载大的核早完成任务，它们的计算资源就会浪费。

第三种类型的协调工作是**同步**。例如，假设需要累加的数据不再是通过计算给出，而是从标准输入（stdin）中读取数据。设 x 是一个数组，存放被 master 核读入的数据：

```
if (I'm the master core)
   for (my_i = 0; my_i < n; my_i++)
      scanf("%lf", &x[my_i]);
```

在大多数系统中，核之间不会自动地同步，而是每个核在自己的空间中工作。在这个例子中，在 master 核初始化完 x 数组并使数组能够被其他核访问之前，不希望其他核开始工作。因此其他核在开始计算部分和之前，需要等待：

```
for (my_i = my_first_i; my_i < my_last_i; my_i++)
   my_sum += x[my_i];
```

需要在初始化 x 数组和计算部分和之间加入一个同步点：

```
Synchronize_cores();
```

7

这里的想法是，每个核会在 Synchronize_cores 函数处等待，直到所有的核都进入该函数——特别是，必须要等到 master 核进入该函数。

目前，功能最强大的并行程序是通过显式的并行结构来编写的，即用扩展 C 或者扩展C++编写的。这些程序包含了显式的并行指令：0 号核执行0 号任务，1 号核执行1 号任务……所有核之间的同步等，因而这种程序通常很复杂。此外，新一代核的复杂性导致即使编写只在单个核上运行的代码，也要特别注意。

当然，仍然可以有其他方法来编写并行程序（如更高级的语言），但这种高层次的开发语言更趋向于牺牲部分性能来降低开发的难度。

## 1.5　我们将做什么

接下来，我们将会学习如何编写显式并行的程序。我们的目标是：学会利用 C 语言和 C 语言的三个不同扩展：**消息传递接口**（Message-Passing Interface，MPI）、**POSIX 线程**（POSIX threads，Pthreads）和 **OpenMP** 来编写基本的并行程序。MPI 和 Pthreads 是 C 语言的扩展库，可以在 C 程序中使用扩展的类型定义、函数和宏；而 OpenMP 包含了一个扩展库以及对 C 编译器的部分修改。

你可能会疑惑：为什么我们需要学习 C 语言的三种不同的扩展而不是一种？问题的答案与这三种扩展以及并行系统都相关。我们要关注的两种主要并行系统是：**共享内存**系统和**分布式内存**

系统。在共享内存系统中，各个核能够共享访问计算机的内存，理论上每个核能够读、写内存的所有区域。因此可以通过检测和更新共享内存中的数据来协调各个核。相反，在分布式内存系统中，每个核都拥有自己的私有内存，核之间的通信是显式的，必须使用类似于在网络中发送消息的机制。图 1-2 给出了两种系统的示意图。Pthreads 和 OpenMP 是为共享内存系统的编程而设计的，它们提供访问共享内存的机制；而 MPI 是为分布式内存系统的编程而设计的，它提供发送消息的机制。

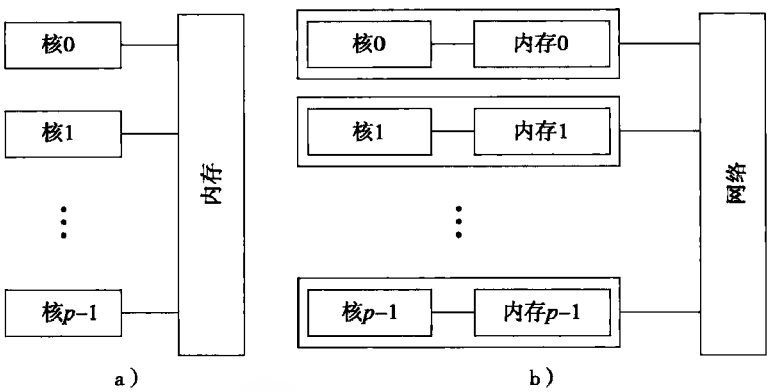



图 1-2 a）共享内存系统；b）分布式内存系统

但是，为什么共享内存系统的编程方法有两种扩展？这是因为，OpenMP 是对 C 语言相对更
[8] 高层次的扩展。例如，它能够通过使用一个简单的指导语句

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

将累加和循环并行化，而 Pthreads 需要做与例子中相似的操作。另一方面，Pthreads 提供了一些在 OpenMP 中不可用的协调构造。OpenMP 容易将很多程序并行化，而 Pthreads 提供的一些构造，增强了并行化其他一些程序的能力。

## 1.6 并发、并行、分布式

如果你看过其他的有关并行计算的书籍，或者在网上搜索与并行计算相关的信息，那么你很可能会遇到如下两个名词：**并发计算**（cocurrent computing）和**分布式计算**（distributed computing）。尽管还没有对并行（parallel）、分布式（distributed）、并发（cocurrent）之间的差别形成统一的意见，但许多作者都同意它们之间有如下区别：

- 在并发计算中，一个程序的多个任务在同一个时段内可以同时执行 [4]。
- 在并行计算中，一个程序通过多个任务紧密协作来解决某个问题。
- 在分布式计算中，一个程序需要与其他程序协作来解决某个问题。

所以并行程序和分布式程序都是并发的，但某些程序如多任务操作系统也是并发的，因为即使它运行在单核机器上，多个任务也能在同一段时间里同时执行。在并行程序和分布式程序之间没有一条明确的分界线，但是并行程序往往同时在多个核上执行多个任务，这些核在物理上紧密
[9] 靠近，或者共享内存或者通过高速网络相互连接。另一方面，分布式程序往往更加“松耦合”。任务是在多个计算机上执行，这些计算机之间相隔较远并且任务是由独立创建的程序来完成的。举例来说，我们前面的两个计算数据全局和的程序是并行的，而网络搜索程序是分布式的。

但需要注意的是，这些术语之间的区别并没有通用的约定。例如，许多作者将共享内存程序看做“并行”的，而将分布式内存程序看做“分布式”的。正如书名所暗示的，本书的兴趣在于并行程序，即紧耦合的多个任务协作来解决某个问题的程序。

## 1.7　本书的其余部分

如何利用本书来帮助我们编写并行程序？

首先，如果你对高性能感兴趣，不管编写串行程序还是并行程序，你都需要对系统的硬件和软件有所了解。在第 2 章中，我们会给出并行软件和硬件的一个回顾。为了理解这个话题，有必要对串行软件和硬件进行回顾。第 2 章给出的资料中，有些在刚开始时是不一定需要了解的，所以可以略过一些，并在阅读后面的章节时再偶尔回过头来参阅。

本书的核心部分是第 3 章到第 6 章，第 3、4、5 章分别给出使用扩展 C 语言的 MPI、Pthreads 和 OpenMP 来编写并行程序的非常基本的介绍。阅读这些章节的前提是对 C 编程语言有所了解。我们尽力使这些章节相互独立，因此可以按任何顺序来阅读。但是，为了使得章节之间相互独立，有些重复是有必要的。所以读完这三章中的任意一章时，在读其他两章时可以适当地略去一些重复的内容。

第 6 章将总结前面章节中所学到的知识，并且开发两个大型的分别面向共享内存系统和分布式内存系统的程序。即便只读完第 3、4、5 章中的一章，也能够理解这两个程序。第 7 章提出了一些对进一步研究并行编程的建议。

## 1.8　警告

在开始之前，我们先提醒一下，仅仅凭感觉来编写并行程序虽然很诱人，但是不进行细致的设计和增量式地开发程序，肯定会犯错。每个并行程序包含至少一个串行程序。因为需要协调多个核的行为，所以编写并行程序肯定比串行程序更加复杂，实际上，是非常复杂。因此，所有设计和开发并行程序的原则都远比开发串行程序的原则重要。 [10]

## 1.9　字体约定

我们将在文中使用如下字体：

- 程序、显示或者运行内容使用代码体。

```
/*  This is a short program */
#include <stdio.h>

int main(int argc, char* argv[]) {
   printf("hello, world\n");

   return 0;
}
```

- 定义是在正文中给出的，定义的术语将用黑体标识。例如：一个**并行程序**能够利用多个核。
- 当需要提及程序开发的环境时，我们一般指的是 UNIX 的 shell，并且用 $ 表示 shell 的提示符：

```
$ gcc -g -Wall -o hello hello.c
```

- 函数调用的语法是包含一个样本参数列表的固定参数列表。例如，求整数绝对值函数 abs，在 stdlib 库中有如下语法格式：

```
int abs(int x);  /* Returns absolute value of int x */
```

对于更复杂的语法结构，我们使用尖括号 < >表示必需的内容，而方括号 [ ] 表示可选的内容，例如 C 中的 if 语句有如下的语法格式：

```
if ( <expression> )
   <statement1>
[else
   <statement2>]
```

这说明，if 语句必须有一个在括号中的语句，并且在右括号后面需要跟一个语句。这个语句可以跟一个 else 子句，也可以不跟。但是如果跟了 else，那么在 else 后面还要跟一
11 个语句。

## 1.10 小结

多年来，我们一直享受着处理器速度不断加快带来的成果。但是，由于物理上的限制，传统处理器性能提升的速度不断降低。为了提高处理器的能力，芯片制造商开始转向**多核**集成电路，即在单块芯片上有多个传统处理器的集成电路。

普通的**串行**程序是指面向单核处理器的、不用利用多核性能的程序。**并行程序**是需要利用多个核的程序。翻译程序不可能担负起将所有的串行程序**并行化**的重任。作为软件开发人员，我们需要学习编写并行程序。

在编写并行程序时，我们需要**协调**各个核的工作。这涉及核之间的**通信、负载平衡**以及**同步**。

在本书中，我们将学习如何编写并行程序，从而最大化程序的性能。我们将使用 C 语言和 MPI、Pthreads 以及 OpenMP。MPI 用于**分布式内存**系统的编程，而 Pthreads 和 OpenMP 用于**共享内存**系统的编程。在分布式内存系统中，每个核都拥有自己的私有内存，而在共享内存系统中，原则上每个核能访问每个内存区域。

**并发**程序是指多个任务在一段时间内同时执行，而**并行**程序和**分布式**程序的多个任务能在同一时刻一起执行。并行程序和分布式程序之间没有非常明确的区别，只是在并行程序中，多个任务是相对紧密耦合的。

并行程序一般是比较复杂的，所以使用好的技术来开发并行程序是非常重要的。

## 1.11 习题

1.1 为求全局总和例子中的 my_first_i 和 my_last_i 推导一个公式。需要注意的是：在循环中，应该给各个核分配数目大致相同的计算元素。*提示*：先考虑 $n$ 能被 $p$ 整除的情况。

1.2 我们已经隐含地假设每次调用 Compute_next_value 函数所执行的工作量与其他次调用 Compute_next_value 函数执行的工作量大致相同。但是，如果当 $i=k$ 时调用这个函数的时间是当 $i=0$ 时调用这个函数所花时间的 $k+1$ 倍，即如果第一次（$i=0$）调用需要 2 毫秒，第二次（$i=1$）调用需要 4 毫秒，第三次（$i=2$）调用需要 6 毫秒，以次类推，那么对于问题 1.1，你该如何回答？

12 1.3 尝试写出图 1-1 中树形结构求全局总和的伪代码。假设核的数目是 2 的幂（1、2、4、8 等）。*提示*：使用变量 divisor 来决定一个核应该是发送部分和还是接收部分和，divisor 的初始值为 2，并且每次迭代后增倍。使用变量 core_difference 来决定哪个核与当前核合作，它的初始值为 1，并且每次迭代后增倍。例如，在第一次迭代中 0 % divisor = 0，1 % divisor = 1，所以 0 号核负责接收和，1 号核负责发送。在第 -次迭代中 0 + core_difference = 1，1 -core_difference = 0，所以 0 号核与 1 号核在第一次迭代中合作。

1.4 作为前面问题的另一种解法，可以使用 C 语言的位操作来实现树形结构的求全局总和。为了了解它是

如何工作的，写下核的二进制数编号是非常有帮助的，注意每个阶段相互合作的成对的核。

| 核 | 阶段 | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| $0_{10}=000_2$ | $1_{10}=001_2$ | $2_{10}=010_2$ | $4_{10}=100_2$ |
| $1_{10}=001_2$ | $0_{10}=000_2$ | × | × |
| $2_{10}=010_2$ | $3_{10}=011_2$ | $0_{10}=000_2$ | × |
| $3_{10}=011_2$ | $2_{10}=010_2$ | × | × |
| $4_{10}=100_2$ | $5_{10}=101_2$ | $6_{10}=110_2$ | $0_{10}=000_2$ |
| $5_{10}=101_2$ | $4_{10}=100_2$ | × | × |
| $6_{10}=110_2$ | $7_{10}=111_2$ | $4_{10}=100_2$ | × |
| $7_{10}=111_2$ | $6_{10}=110_2$ | × | × |

从表中我们可以看到在第一阶段，每个核与其二进制编号的最右位不同编号的核配对，在第二阶段与其二进制编号的最右第二位不同编号的核配对，第三阶段与其二进制编号的最右第三位不同编号的核配对。因此，如果在第一阶段有二进制掩码（bit mask）$001_2$、第二阶段有 $010_2$、第三阶段有 $100_2$，那么可以通过将编号中对应掩码中非零位置的二进制数取反来获得配对核的编号，也即通过异或操作或者^操作。

使用位异或操作或者左移操作编写伪代码来实现上述算法。

1.5　如果核的数目不是 2 的幂（例如 3、5、6、7），那么在习题 1.3 或者习题 1.4 中编写的伪代码还能运行吗？修改伪代码，使得在核数目未知的情况下仍然能运行。

1.6　在下列情况中，推导公式求出 0 号核执行接收与加法操作的次数。

a. 最初的求全局总和的伪代码。

b. 树形结构求全局总和。 [13]

制作一张表来比较这两种算法在总核数是 2、4、8、…、1024 时，0 号核执行的接收与加法操作的次数。

1.7　全局总和例子中的第一部分（每个核对分配给它的计算值求和），通常认为是数据并行的例子；而第一个求全局总和例子的第二个部分（各个核将它们计算出的部分和发送给 master 核，master 核将这些部分和再累加求和），认为是任务并行。第二个全局和例子中的第二部分（各个核使用树形结构累加它们的部分和），是数据并行的例子还是任务并行的例子？为什么？

1.8　假如系里的老师要为学生举办一个聚会。

a. 在准备聚会的时候，如何分配各种任务给各个老师，以实现任务并行？设计一个方案使得各种任务能够同时进行。

b. 我们希望其中的一项任务是清理聚会场地，那么该如何分配清扫任务以实现数据并行？

c. 设计一个任务并行和数据并行相结合的方案来准备聚会（如果教师的工作量太大，可以让助教来帮忙）。

1.9　写一篇文章来描述你研究方向中因使用并行计算而获益的事。大致地描述是如何使用并行的。你将用到任务并行还是数据并行？ [14]

第2章
An Introduction to Parallel Programming

# 并行硬件和并行软件

让学科专家而不是计算机科学与计算机工程专家编写并行程序是完全可行的。但是，为了编写高效的并行程序，我们需要对底层的硬件和系统软件有所了解。理解不同类型并行软件的相关知识是有用的，所以，本章将简要介绍硬件和软件方面的一些知识，也会简略介绍如何评价程序性能以及开发并行程序的方法。在本章的末尾，我们将介绍在本书中其余部分涉及的开发环境、规则以及一些假设。

本章篇幅相对比较长，内容涵盖也比较广泛。在首次阅读时，你可以略过细节，通过浏览了解本章的内容。在阅读后面的章节时，如果对某个概念或者名词不清楚时，可以再回到本章查询。特别地，在2.2节中除2.2.1节外的大部分材料，以及2.3.1节和2.3.3节的内容，都可以先跳过。

## 2.1 背景知识

并行硬件和并行软件是从传统的一次只执行单个任务的**串行**硬件和**串行**软件中发展出来的。所以为了更好地理解并行系统的现状，我们先简略地了解一下串行系统的特性。

### 2.1.1 冯·诺依曼结构

“经典”的**冯·诺依曼**结构包括**主存**、**中央处理单元**（Central Processing Unit，CPU）处理器或核，以及主存和CPU之间的**互连结构**。主存中有许多区域，每个区域都可以存储指令和数据。
[15] 每个区域都有一个地址，可以通过这个地址来访问相应的区域及区域中存储的数据和指令。

中央处理单元分为控制单元和算术逻辑单元（Arithmetic Logic Unit，ALU）。控制单元负责决定应该执行程序中的哪些指令，而ALU负责执行指令。CPU中的数据和程序执行时的状态信息存储在特殊的快速存储介质中，即**寄存器**。控制单元有一个特殊的寄存器，叫做**程序计数器**，用来存放下一条指令的地址。

指令和数据通过CPU和主存之间的互连结构进行传输。这种互连结构通常是**总线**，总线中包括一组并行的线以及控制这些线的硬件。冯·诺依曼机器一次执行一条指令，每条指令对一个数据进行操作。见图2-1。

当数据或指令从主存传送到CPU时，我们称数据或指令从内存中**取出**或者**读出**。当数据或指令从CPU传送到主存中时，我们称数据或指令**写入**或者**存入**内存中。主存和CPU之间的分离称
[16] 为**冯·诺依曼瓶颈**。这是因为互连结构限定了指令和数据访问的速率。程序运行所需要的大部分数据和指令被有效地与CPU隔离开。到2010年，CPU执行指令的速度是从主存中取指令速度的100多倍。

为了更好地理解这个问题，我们可以想象一个大型企业在某个镇上有一个工厂（CPU），在另一个镇上有一个仓库（主存）。在工厂和仓库之间只有一条双车道公路。生产产品所需的原材料都存储在仓库中，所有的制成品在交付给客户前也存储在仓库中。如果产品生产的速度远大于原材料和产品运输的速度，那么就会出现交通堵塞，工厂的工人和机器要么时不时地空闲，要么就降低生产速度。

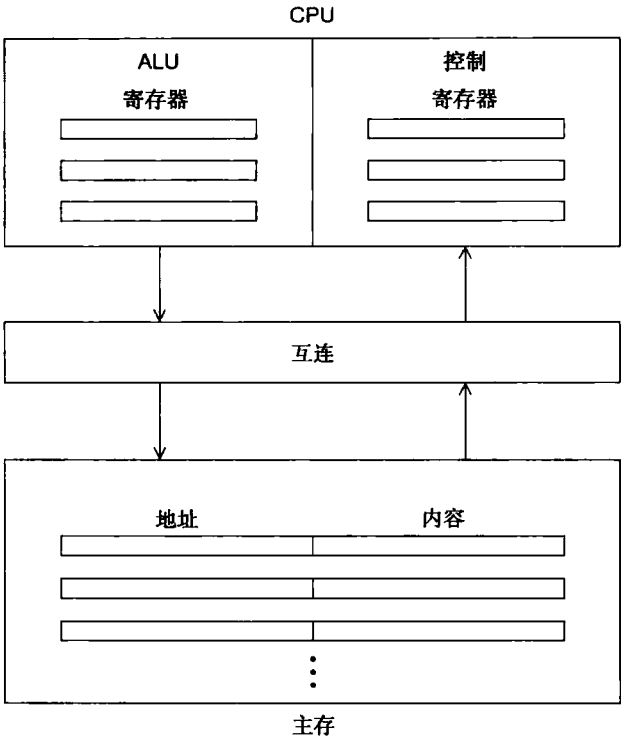



图 2-1　冯 · 诺依曼结构

为了解决冯 · 诺依曼瓶颈，或者更概括地说，提高 CPU 的性能，计算机工程师和科学家已经多次尝试对基本的冯 · 诺依曼结构进行修改。在讨论这些修改前，让我们先花点时间讨论在冯 · 诺依曼系统和现在系统上运行的软件。

### 2.1.2　进程、多任务及线程

**操作系统**（Operating System，OS）一种用来管理计算机的软件和硬件资源的主要软件。它决定什么程序能运行以及什么时候运行。它控制运行中程序的内存分配以及对诸如硬盘、网卡等外设的访问。

当用户运行一个程序时，操作系统创建一个**进程**。进程是运行着的程序的一个实例。一个进程包括如下实体：

- 可执行的机器语言程序。
- 一块内存空间，包括可执行代码，一个用来跟踪执行函数的**调用栈**、一个**堆**，以及一些其他内存区域。
- 操作系统分配给进程的资源描述符，如文件描述符。
- 安全信息，例如阐述进程能够访问哪些硬件和软件的信息。
- 进程状态信息，例如进程是否就绪还是等待某些资源、寄存器内容，以及关于进程存储空间的信息。

大多数现代操作系统都是**多任务**的。这意味着操作系统提供对同时运行多个程序的支持。这 17
对于单核系统也是可行的，因为每个进程只运行一小段时间（几毫秒），亦即一个**时间片**。在一个程序执行了一个时间片的时间后，操作系统就切换执行其他程序。一个多任务操作系统能够在 1 分钟内多次切换运行的程序，即使切换进程花费相对较长的时间。

在一个多任务操作系统中，如果一个进程需要等待某个资源，例如需要从外部的存储器读数

据，它会**阻塞**。这意味着，该进程会停止运行，操作系统可以运行其他进程。但是，许多程序能够继续工作即使当前运行的部分必须等待某些资源。例如：航班预定系统在一个用户因为等待座位图而阻塞时，可为另一个用户提供可用的航线查询。**线程**为程序员提供了一种机制，将程序划分为多个大致独立的任务，当某个任务阻塞时能执行其他任务。此外，在大多数系统中，线程间的切换比进程间的切换更快。因为线程相对于进程而言是“轻量级”的。线程包含在进程中，所以线程可以使用相同的可执行代码，共享相同的内存和相同的 I/O 设备。实际上，当两个线程共属于一个进程时，它们共享进程的大多数资源。它们之间最大的差别是各自需要一个私有的程序计数器和函数调用栈，使它们能够独立运行。

如果进程是执行的“主线程”，其他线程由主线程启动和停止，那么我们可以设想进程和它的子线程如下进行，当一个线程开始时，它从进程中**派生**（fork）出来；当一个线程结束，它**合并**（join）到进程中，如图 2-2 所示。

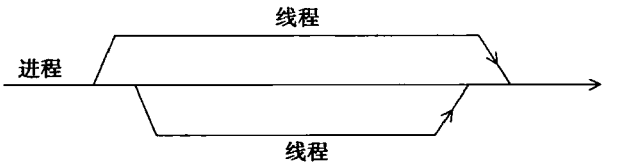



图 2-2　一个进程与两个线程

## 2.2　对冯·诺依曼模型的改进

正如我们前面提到的，第一台电子数字计算机是在 20 世纪 40 年代诞生的，计算机科学家和计算机工程师对基本的冯·诺依曼结构已经做了很多改进。许多改进都是为了解决冯·诺依曼瓶颈，也有许多改进只是为了使 CPU 更快。本节中，我们会看到三种改进措施：缓存（caching）、
[18] 虚拟存储器（或虚拟内存）、低层次并行。

### 2.2.1　Cache 基础知识

缓存是解决冯·诺依曼瓶颈而最广泛使用的方法之一。为了理解缓存背后的思想，我们再来回忆前面的例子。一个大型企业在某个镇上有一个工厂（CPU），在另一个镇上有一个仓库（主存）。在工厂和仓库之间只有一条双车道公路。有许多方法可以解决工厂和仓库之间的原材料和制成品运输难题。其中之一是加宽公路，另一个方法是迁移工厂或者仓库，或者建立一个统一的工厂和仓库。缓存结合了这两种方法。不再是一次传输一条指令或者一个数据，而是将互连通路加宽，使得一次内存访问能够传送更多的数据或者更多的指令。而且，不再是将所有的数据和指令存储在主存中，可以将部分数据块或者代码块存储在一个靠近 CPU 寄存器的特殊存储器里。

一般来说，对**高速缓冲存储器**（cache，简称缓存）的访问时间比其他存储区域的访问时间短。在本书中，当谈到缓存时，一般指的是 **CPU 缓存**（CPU Cache）。CPU Cache 是一组相比于主存，CPU 能更快速地访问的内存区域。CPU Cache 位于与 CPU 同一块的芯片或者位于其他芯片上，但比普通的内存芯片能更快地访问。

有了 Cache 后，一个很明显的问题是什么样的数据和指令能够存储在 Cache 中。通用的准则基于下面的原理：程序接下来可能会用到的指令和数据与最近访问过的指令和数据在物理上是邻近存放的。在执行完一条指令后，程序通常会执行下一条指令。同样，当程序访问一个内存区域后，通常会访问物理位置靠近的下一个区域。一个例子是在使用数组时，考虑下面的循环：

```
float z[1000];
. . .
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

数组在内存中是连续分配的，所以存储 z[1] 数据的内存区域紧接在存储 z[0] 的内存区域后

面。因此，只要 i <999，在读完 z[i] 的数据之后总是立即读 z[i +1] 的数据。

程序访问完一个存储区域往往会访问接下来的区域，这个原理称为**局部性**。在访问完一个内存区域（指令或者数据），程序会在不久的将来（**时间**局部性）访问邻近的区域（**空间**局部性）。

为了运用局部性原理，系统使用更宽的互连结构来访问数据和指令。也就是：一次内存访问能存取一整块代码和数据，而不只是单条指令和单条数据。这些块称为**高速缓存块**或者**高速缓存行**。一个典型的高速缓存行能存储 8 ~ 16 倍单个内存区域的信息。在我们的例子中，如果一个高 [19] 速缓存行可以存放 16 个浮点数，当执行 sum + =z[0] 时，系统可能把数组 z 最开始的 16 个元素：z[0]、z[1]、…、z[15] 从主存读到 Cache 中。因此，在后面的 15 次加法运算中，需要使用的数据已经在 Cache 中。

从概念上，很容易把 CPU Cache 认为是单一结构，但实际上，Cache 分为不同的**层**（level）。第一层（L1）最小但最快，更高层 Cache（L2、L3、…）更大但相对较慢。2010 年，大多数系统拥有至少两层 Cache，有三层 Cache 是非常普遍的。Cache 通常是用来存储速度较慢的存储器中信息的副本，可以认为低层 Cache（更快、更小）是高层 Cache 的 Cache。所以，一个变量存储在 L1 Cache 中，也会存储在 L2 Cache 中。但是，有些多层 Cache 不会复制已经在其他层 Cache 中存在的信息。对于这种 Cache，L1 Cache 中的变量不会存储在其他层 Cache 中，但会存储在主存中。

当 CPU 需要访问指令或者数据时，它会沿着 Cache 的层次结构向下查询：首先查询 L1 Cache，接着 L2 Cache，以此类推。最后，如果 Cache 中没有所需要的信息，就会访问主存。当向 Cache 查询信息时，如果 Cache 中有信息，则称为 **Cache 命中**或者**命中**；如果信息不存在，则称为 **Cache 缺失**或者**缺失**。命中和缺失是相对 Cache 层而言的。例如，当 CPU 试图访问某个变量时，很可能 L1 Cache 缺失，而 L2 Cache 命中。

注意，存储器访问的术语**读**（read）和**写**（write）也适用于 Cache，例如我们可以从 L2 Cache 中读一条指令，也可以向 L1 Cache 写数据。

当 CPU 尝试读数据或者指令时，如果发生 Cache 缺失，那么就会从主存中读出包含所需信息的整个高速缓存块。这时 CPU 会阻塞，因为它需要等待速度相对较慢的主存：处理器可以停止执行当前程序的指令，直到从主存中取出所需的数据或者指令。例如，当我们读 z[0] 时，处理器会阻塞直到包括 z[0] 的高速缓存块从主存传送到 Cache 中。

当 CPU 向 Cache 中写数据时，Cache 中的值与主存中的值就会不同或者**不一致**（inconsistent）。有两种方法来解决这个不一致性问题。在**写直达**（write-through）Cache 中，当 CPU 向 Cache 写数据时，高速缓存行会立即写入主存中。在**写回**（write-back）Cache 中，数据不是立即更新到主存中，而是将发生数据更新的高速缓存行标记成脏（dirty）。当发生高速缓存行替换时，标记为脏的高速缓存行被写入主存中。

### 2.2.2 Cache 映射

在 Cache 设计中，另一个问题是高速缓存行应该存储在什么位置。当从主存中取出一个高速缓存行时，应该把这个高速缓存行放置到 Cache 中的什么位置？这个问题的答案因系统而异。一个极端是**全相联**（fully associative）Cache，每个高速缓存行能够放置在 Cache 中的任意位置。另 [20] 一个极端是**直接映射**（directed mapped）Cache，每个高速缓存行在 Cache 中有唯一的位置。处于两种极端中间的方案是 ***n* 路组相联**（n-way set associated）。在 *n* 路组相联 Cache 中，每个高速缓存行都能放置到 Cache 中 *n* 个不同区域位置中的一个。例如，在 2 路组相联 Cache 中，每个高速缓存行可以映射到 2 个位置中的一个。

假设主存有 16 行，分别用 0 ~ 15 标记，Cache 有 4 行，用 0 ~ 3 标记。在全相联映射中，主存中的 0 号行能够映射到 Cache 中的 0、1、2、3 任意一行。在直接映射 Cache 中，可以根据主存中

高速缓存行的标记值除以 4 求余，获得在 Cache 中的索引。因此主存中 0、4、8 号行会映射到 Cache 的 0 号行，主存中的 1、5、9 号行映射到 Cache 的 1 号行，以此类推。在 2 路组相联 Cache 中，将 Cache 分成两组，0 号行和 1 号行构成一个组，称为 0 号组；2 号行和 3 号行构成另一个组，称为 1 号组。根据主存中行的标记对 2 取模从而获得 Cache 中组的索引号。主存的 0 号行可以映射到 Cache 中第 0 组中的 0 号行或者 1 号行。详见表 2-1。

**表 2-1　将 16 行的主存映射到 4 行的 Cache 上**

| 内存索引 | 高速缓存中的位置 | | |
|---|---|---|---|
| | 全相联 | 直接映射 | 2 路组相联 |
| 0 | 0，1，2 或 3 | 0 | 0 或 1 |
| 1 | 0，1，2 或 3 | 1 | 2 或 3 |
| 2 | 0，1，2 或 3 | 2 | 0 或 1 |
| 3 | 0，1，2 或 3 | 3 | 2 或 3 |
| 4 | 0，1，2 或 3 | 0 | 0 或 1 |
| 5 | 0，1，2 或 3 | 1 | 2 或 3 |
| 6 | 0，1，2 或 3 | 2 | 0 或 1 |
| 7 | 0，1，2 或 3 | 3 | 2 或 3 |
| 8 | 0，1，2 或 3 | 0 | 0 或 1 |
| 9 | 0，1，2 或 3 | 1 | 2 或 3 |
| 10 | 0，1，2 或 3 | 2 | 0 或 1 |
| 11 | 0，1，2 或 3 | 3 | 2 或 3 |
| 12 | 0，1，2 或 3 | 0 | 0 或 1 |
| 13 | 0，1，2 或 3 | 1 | 2 或 3 |
| 14 | 0，1，2 或 3 | 2 | 0 或 1 |
| 15 | 0，1，2 或 3 | 3 | 2 或 3 |

当内存中的行（多于一行）能被映射到 Cache 中的多个不同位置（全相联和 $n$ 路组相联）时，需要决定替换或者**驱逐** Cache 中的哪一行。在前面的 2 路组相联的例子中，假设主存中的 0 号行已存储在 Cache 的 0 号行，主存中的 2 号行已存储在 Cache 的 1 号行，那么主存的 4 号行
[21] 应该存储在哪里呢？最常用的替换方案是**最近最少使用**（least recently used）。顾名思义，Cache 记录各个块被访问的次数，替换最近访问次数最少的块。如果近来主存的 0 号行比 2 号行访问的更多，那么 2 号行在就会被替换出 Cache，它在原来 Cache 的位置就被替换成用来存储 4 号主存行。

### 2.2.3　Cache 和程序：一个实例

非常重要的一点是：CPU Cache 是由系统硬件来控制的，而编程人员并不能直接决定什么数据和什么指令应该在 Cache 中。但是，了解空间局部性和时间局部性原理可以让我们对 Cache 有些许间接的控制。例如，C 语言以“行主序”来存储二维数组。尽管二维数组看上去是一个矩形块，但是内存是巨大的一维数组。在行主序存储模式下，先存储二维数组的第一行，接着第二行，以次类推。下面的两段代码中，第一个嵌套循环比第二个嵌套循环有更好的性能，因为它顺序访问二维数组中的数据。

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

为了更好地理解，假设 MAX 等于4，那么数组 A 中元素的存储位置为：

| 高速缓存行 | 数组 A 中的元素 | | | |
|---|---|---|---|---|
| 0 | A [0] [0] | A [0] [1] | A [0] [2] | A [0] [3] |
| 1 | A [1] [0] | A [1] [1] | A [1] [2] | A [1] [3] |
| 2 | A [2] [0] | A [2] [1] | A [2] [2] | A [2] [3] |
| 3 | A [3] [0] | A [3] [1] | A [3] [2] | A [3] [3] |

所以，A[0][1]存储在A[0][0]的后面，而A[1][0]存储在A[0][3]的后面。

假设刚开始时，Cache 中没有 A 数组的任何元素，一个高速缓存行可以存放 A 的4个元素，并且 A[0][0]是高速缓存行中的第一个元素。最后，假设 Cache 是直接映射的，只能存储 A 数组的8个元素，即两个高速缓存行（我们不关注 x 和 y）。 22

两个循环都尝试首先访问 A[0][0]，因为它不在 Cache 中，所以这将导致一次 Cache 缺失，然后系统将包含 A[0][0]、A[0][1]、A[0][2]、A[0][3]的行从内存中读出并写入 Cache 中。第一个循环接下来会依次访问 A[0][1]、A[0][2]、A[0][3]，它们都在 Cache 中，而下一次 Cache 缺失就会发生在代码访问 A[1][0]的时候。按照上面的分析，我们可以看到第一个循环在访问数组 A 的元素时，总共发生4次 Cache 缺失，每行发生一次。值得注意的是，我们假想的 Cache 只能存储两个高速缓存行，即8个元素，当读第三行的第一个元素和第四行的第一个元素时，Cache 中已存在的行必须被替换出去。一旦某行被替换出去，第一个循环就不会再次访问该行的元素。

在将第一行元素读出并写入 Cache 中后，第二个循环需要访问 A[1][0]、A[2][0]、A[3][0]，但是它们都不在 Cache 中。所以下面三次访问 A 都将导致 Cache 缺失。此外，因为 Cache 比较小，读 A[2][0]和A[3][0]会导致 Cache 中原有的行被替换出去。因为 A[2][0]在2号行中，读该行会导致0号高速缓存行被替换出去，而读 A[3][0]会导致1号高速缓存行被替换出去。在执行完一次内部循环后，要访问 A[0][1]，但它原来所在的0号高速缓存行已经被替换出去了。所以我们可以看到，每次读 A 的元素，就会发生一次 Cache 缺失，所以第二个循环总共发生16次缺失。

因此，我们可以预测第一个嵌套循环比第二个运行速度快。在实际运行的时候，当 MAX = 1000，第一个嵌套循环的运行速度近3倍快于第二个嵌套循环。

### 2.2.4 虚拟存储器

Cache 使得 CPU 快速访问主存中的指令和数据变成可能。但是，如果运行一个大型的程序，或者程序需要访问大型数据集，那么所有的指令或者数据可能在主存中放不下。这种情况在多任

务操作系统中时常发生；为了在程序间切换并且造成一种多个程序能够同时运行的错觉，下一个时间片运行所需的指令和数据必须在主存中。因此，在多任务操作系统中，即使主存非常大，许多运行中的程序必须共享可用的主存。此外，这种共享必须确保每个程序的数据和指令能被保护，不会被其他程序访问。

利用**虚拟存储器**（或**虚拟内存**），使得主存可以作为辅存的缓存。它通过在主存中存放当前
[23] 执行程序所需要用到的部分，来利用时间和空间局部性；那些暂时用不到的部分存储在辅存的块中，称为**交换空间**（swap space）中。与 CPU Cache 类似，虚拟存储器也是对数据块和指令块进行操作。这些块通常称为**页**（page）。因为访问辅存比访问内存要慢几十万倍，所以页通常比较大。大多数系统采用固定大小的页，从 4 ~ 16kB 不等。

如果在编译程序时直接给页指定物理内存地址，那么就会陷入麻烦。因为这样做的话，程序中的每一页也都必须指定一块内存来存放，在多任务操作系统中，就会导致多个程序要使用相同的内存块。为了避免这个问题，在编译程序时，给程序的页赋予虚拟页号。当程序运行时，创建一张将虚拟页号映射成物理地址的表。程序运行时使用到虚拟地址，这个**页表**就用来将虚拟地址转换成物理地址。假如页表的创建由操作系统管理，那么就能保证程序使用的内存不会与其他程序使用的内存重叠。

使用页表的缺点是，会使访问主存区域的时间加倍。假设，想要执行主存中的一条指令，执行程序只有该指令的虚拟地址，在从主存中找到该指令前，需要将虚拟地址转换成物理地址。为了能够转换虚拟地址，需要在内存中寻找包含该指令的页。现在，虚拟页号作为虚拟地址的一部分存储。假如虚拟地址有 32 位，页大小是 4KB = 4096 字节，那么可以用 12 位来标识页中的每个字节。这是因为 $2^{12}=4096$。因此，可以用虚拟地址的低 12 位来定位页内字节，而剩下的位用来定位页。详见表 2-2。虚拟页号能够直接从虚拟地址中计算出来，而不用访存。但是，一旦知道了虚拟页号，就需要访问页表，将虚拟页号转换成物理页号。如果所需要的页表不在 Cache 中，就需要将它从内存中加载到 Cache 中。加载结束后，就能将虚拟地址转换成物理地址并获取需要的指令。

**表 2-2　虚拟地址分为两部分：虚拟页号和页内字节偏移量**

| 虚拟地址 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 虚拟页号 | | | | | 页内字节偏移量 | | | | |
| 31 | 30 | … | 13 | 12 | 11 | 10 | … | 1 | 0 |
| 1 | 0 | … | 1 | 1 | 0 | 0 | … | 1 | 1 |

[24] 显然地，问题又来了。尽管多个程序可以同时使用主存了，但使用页表会增加程序总体的运行时间。为了解决这个问题，处理器有一种专门用于地址转换的缓存，叫做**转译后备缓冲区**（Translation-Lookaside Buffer，TLB）。TLB 在快速存储介质中缓存了一些页表的条目（通常为 16 ~ 512 条）。利用时间和空间局部性原理，大部分存储器所访问页的物理地址已经存储在 TLB 中，对主存中页表的访问能够大幅度减少。

TLB 的术语和 Cache 的术语一样。当查询的地址和虚拟页号在 TLB 中时，称为 TLB **命中**；如果不在 TLB 中，称为 TLB **缺失**。但是，有些术语也不同，假如想要访问的页不在内存中，即页表中该页没有合法的物理地址，该页只存储在磁盘上，那么这次访问称为**页面失效**（page fault）。

磁盘访问的相对迟缓也会给虚拟内存带来一些额外的后果。第一，在 CPU Cache 中，可以使用写直达或者写回方案来处理写缺失问题，但在虚拟内存中，磁盘访问代价太大，需要尽可能地避免，所以虚拟内存通常采用写回方案。可以在内存中为每个页设置一位，标识该页是否被更新

过。如果该页被更新过，则在页从内存中替换出去时，需要把它写入磁盘。第二，因为磁盘访问迟缓，管理页表和处理磁盘访问由操作系统来完成。因此，程序员不能直接控制虚拟内存。CPU Cache是由系统硬件控制，而虚拟内存是由系统硬件和操作系统一起控制的。

### 2.2.5 指令级并行

**指令级并行**（Instruction-Level parallelism，ILP）通过让多个处理器部件或者**功能单元**同时执行指令来提高处理器的性能。有两种主要方法来实现指令级并行：**流水线**和**多发射**。**流水线**是指将功能单元分阶段安排；**多发射**是指让多条指令同时启动。这两种方法在现代CPU中都有使用。

**流水线**

流水线的原理与工厂的装配流水线类似：一个小组将汽车的引擎栓到底盘上的同时，另一个小组为第一个小组已处理过的部件连接变速器、传动轴和引擎，与此同时第三个小组把前面两个小组已完成的产品装上车架。举一个关于计算的例子，假如我们想要将浮点数 $9.87 \times 10^4$ 和 $6.54 \times 10^3$ 相加，我们可以使用如下步骤：

25

| 时 间 | 操 作 | 操作数一 | 操作数二 | 结 果 |
|---|---|---|---|---|
| 0 | 取操作数 | $9.87 \times 10^4$ | $6.54 \times 10^3$ | |
| 1 | 比较指数 | $9.87 \times 10^4$ | $6.54 \times 10^3$ | |
| 2 | 移位一个操作数 | $9.87 \times 10^4$ | $0.654 \times 10^4$ | |
| 3 | 相加 | $9.87 \times 10^4$ | $0.654 \times 10^4$ | $10.524 \times 10^4$ |
| 4 | 规格化结果 | $9.87 \times 10^4$ | $0.654 \times 10^4$ | $1.052\,4 \times 10^5$ |
| 5 | 舍入结果 | $9.87 \times 10^4$ | $0.654 \times 10^4$ | $1.05 \times 10^5$ |
| 6 | 存储结果 | $9.87 \times 10^4$ | $0.654 \times 10^4$ | $1.05 \times 10^5$ |

这里，我们使用的数是以10为基数，三位尾数或者其中一位尾数表示在小数点的左边。因此，在这个例子中，规格化操作将小数点向左移动一个单位，并将尾数最终舍入成三位数字。

现在，如果每次操作花费1纳秒（$10^{-9}$秒），那么加法操作需要花费7纳秒。所以，如果执行如下的代码：

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

那么for循环需要花费7000纳秒。

还有另一个方案，将浮点数加法器划分成7个独立的硬件或者功能单元。第一个单元取两个操作数，第二个比较指数，以此类推。假设一个功能单元的输出是下面一个功能单元的输入，那么加法功能单元的输出是规格化结果功能单元的输入。一次浮点数加法花费7纳秒时间，但是，当执行for循环时，可以在比较x[0]和y[0]指数时取出x[1]和y[1]。更一般地说，能够同时执行7条指令的7个不同阶段。详见表2-3。从表2-3中可以看到，在时间5后，流水循环每1纳秒产生一个结果，而不再是每7纳秒一次。所以，执行for循环的总时间从7000纳秒降低到1006纳秒，提高了近7倍。

总的来说，$k$ 个阶段的流水线不可能达到 $k$ 倍的性能提高。例如，如果各种功能单元的运行时间不同，则每个阶段的有效运行时间取决于最慢的功能单元。此外，有些延迟（例如等待操作数）也会造成流水线的阻塞。关于流水线的性能，可以参考习题2.1。

26

表 2-3 流水线加法。表格中的数字表示操作数/结果的下标

| 时间 | 取 | 比较 | 移位 | 加 | 规格化 | 四舍五入 | 存储 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | | | | | | |
| 1 | 1 | 0 | | | | | |
| 2 | 2 | 1 | 0 | | | | |
| 3 | 3 | 2 | 1 | 0 | | | |
| 4 | 4 | 3 | 2 | 1 | 0 | | |
| 5 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 6 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 999 | 999 | 998 | 997 | 996 | 995 | 994 | 993 |
| 1000 | | 999 | 998 | 997 | 996 | 995 | 994 |
| 1001 | | | 999 | 998 | 997 | 996 | 994 |
| 1002 | | | | 999 | 998 | 997 | 996 |
| 1003 | | | | | 999 | 998 | 997 |
| 1004 | | | | | | 999 | 998 |
| 1005 | | | | | | | 999 |

### 多发射

流水线通过将功能分成多个单独的硬件或者功能单元，并把它们按顺序串接来提高性能。而多发射处理器通过复制功能单元来同时执行程序中的不同指令。例如，假设有两个完整的浮点数加法器，则计算下面循环所需要的时间减半：

```
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

当第一个加法器计算 z[0]时，第二个加法器计算 z[1]；当第一个加法器计算 z[2]时，第二个计算 z[3]；以次类推。

如果功能单元是在编译时调度的，则称该多发射系统使用**静态**多发射；如果是在运行时间调度的，则称该多发射系统使用**动态**多发射。一个支持动态多发射的处理器称为**超标量**（superscalar）。

当然，为了能够利用多发射，系统必须找出能够同时执行的指令。其中一种最重要的技术是**预测**（speculation）。在预测技术中，编译器或者处理器对一条指令进行猜测，然后在猜测的基础上执行代码。一个简单的例子，在下面的代码中，系统预测 z = x + y 的结果 z 可能为正数，因
27 此执行赋值操作 w = x。

```
z = x + y;
if (z > 0)
   w = x;
else
   w = y;
```

另一个例子，在代码中：

```
z = x + y;
w = *a_p;  /* a_p is a pointer */
```

系统可能预测指针 a_p 不指向 z，因此能够同时执行代码中两个赋值操作。

正如以上两个例子所解释的那样，预测执行允许预测错误的情况发生。在第一个例子中，如果 z = x + y 的值为负或者为零，需要回退机制，然后执行 w = y。在第二个例子中，假如 a_p 指向 z，则需要重新执行赋值操作 w = * a_p。

如果预测工作由编译器来做，那么它通常在代码中嵌入测试语句来验证预测的正确性，如果预测错误，就会执行修正操作。假如由硬件做预测操作，处理器一般会将预测执行的结果缓存在一个缓冲器中。如果预测正确，缓冲器中的内容会传递给寄存器或者内存；如果预测错误，则缓冲器中的内容被丢弃，指令重新执行。

尽管动态多发时系统能够乱序执行指令，但在现行的系统中，指令是顺序加载的，执行的结果也是顺序提交的。即指令的结果是按程序中规定的顺序写入寄存器和内存中的。

另一方面，编译器优化技术能够对指令进行重新排序。我们会在后面看到，这一操作会对共享内存的编程产生重大影响。

### 2.2.6　硬件多线程

指令级并行是很难利用的，因为程序中有许多部分之间存在依赖关系。例如，直接计算斐波那契数：

```
f[0] = f[1] = 1;
for (i = 2; i <= n; i++)
   f[i] = f[i-1] + f[i-2];
```

在上述代码中，实质上根本没有可以同时执行的指令。

**线程级并行**（Thread-Level Parallelism，TLP）尝试通过同时执行不同线程来提供并行性。与 ILP 相比，TLP 提供的是**粗粒度**的并行性，即同时执行的程序基本单元（线程）比**细粒度**的程序单元（单条指令）更大或者更粗。 28

**硬件多线程**（hardware multithreading）为系统提供了一种机制，使得当前执行的任务被阻塞时，系统能够继续其他有用的工作。例如，如果当前任务需要等待数据从内存中读出，那么它可以通过执行其他线程而不是继续当前线程来发掘并行性。当然，为了使这种机制有效，系统必须支持线程间的快速切换。例如，在一些较老的系统中，一个线程被简单地实现为一个进程，但进程之间切换的时间是执行指令时间的数千倍。

在**细粒度**（fine-grained）多线程中，处理器在每条指令执行完后切换线程，从而跳过被阻塞的线程。尽管这种方法能够避免因为阻塞而导致机器时间的浪费，但它的缺点是，执行很长一段指令的线程在执行每条指令的时候都需要等待。**粗粒度**（coarse-grained）多线程为了避免这个问题，只切换那些需要等待较长时间才能完成操作（如从主存中加载）而被阻塞的线程。这种机制的优点是，不需要线程间的立即切换。但是，处理器还是可能在短阻塞时空闲，线程间的切换也还是会导致延迟。

**同步多线程**（Simultaneous Multithreading，SMT）是细粒度多线程的变种。它通过允许多个线程同时使用多个功能单元来利用超标量处理器的性能。如果我们指定“优先”线程，那么能够在一定程度上减轻线程减速的问题。优先线程是指有多条指令就绪的线程。

## 2.3　并行硬件

因为有多个复制的功能单元，所以多发射和流水线可以认为是并行硬件。但是，这种并行性通常对程序员是不可见的，所以我们仍把它们当成基本的冯 · 诺依曼结构的扩展。我们的原则是，并行硬件应该是仅限于对程序员可见的硬件。换句话说，如果能够通过修改源代码而开发并行性或者必须修改源代码来开发并行性，那么我们认为这种硬件是并行硬件。

### 1.3.1 SIMD系统

在并行计算中，Flynn 分类法经常用来对计算机体系结构进行分类。按照它能够同时管理的指令流数目和数据流数目来对系统分类。因此典型的冯·诺依曼系统是**单指令流单数据流**（Single Instruction Stream，Single Data Stream，SISD）系统，因为它一次执行一条指令，一次存取一个数据项。

**单指令多数据流**（Single Instruction，Multiple Data，SIMD）系统是并行系统。顾名思义，
29 SIMD 系统通过对多个数据执行相同的指令从而实现在多个数据流上的操作。所以一个抽象的 SIMD 系统可以认为有一个控制单元和多个 ALU。一条指令从控制单元广播到多个 ALU，每个 ALU 或者在当前数据上执行指令或者处于空闲状态。例如，假设想要执行一个“向量加法”，即有两个数组 x 和 y，每个都有 $n$ 个元素，想要把 y 中的元素加到 x 中：

```
for (i = 0; i < n; i++)
   x[i] += y[i];
```

假如 SIMD 系统中有 $n$ 个 ALU，我们能够将 x[i]和 y[i]加载到第 $i$ 个 ALU 中，然后让第 $i$ 个 ALU 将 x[i]和 y[i]相加，最后将结果存储在 x[i]中。如果系统有 $m$ 个 ALU，并且 $m < n$，那么能够一次同时执行 $m$ 个元素的加法。例如，假设 $m = 4$，$n = 15$，可以先将编号为 0 ~ 3 的 x 和 y 的元素相加，接着是 4 ~ 7 的元素，然后是 8 ~ 11，最后是 12 ~ 14 的元素相加。注意，我们例子中最后一组元素（元素 12 ~ 14）的加法，只对 x 和 y 的三个元素操作，所以 4 个 ALU 中的一个会处于空闲状态。

所有的 ALU 要么执行相同的指令，要么同时处于空闲状态的要求会严重地降低 SIMD 系统的整体性能。例如，如果只想在 y[i]大于 0 时才执行加法操作：

```
for (i = 0; i < n; i++)
   if (y[i] > 0.0) x[i] += y[i];
```

在这种情况下，我们必须将 y 的每一个元素加载到一个 ALU 中，然后判断是否为正。如果 y[i]为正，就继续执行加法操作；否则，加载了 y[i]的 ALU 处于空闲状态，而其他的 ALU 执行加法。

注意，在“经典”的 SIMD 系统中，ALU 必须同步操作，即在下一条指令开始执行之前，每个 ALU 必须等待广播。此外，ALU 没有指令存储器，所以 ALU 不能通过存储指令来延迟执行指令。

最后，正如我们第一个例子中显示的，SIMD 系统适合于对处理大型数组的简单循环实行并行化。通过将数据分配给多个处理器，然后让各个处理器使用相同的指令来操作数据子集实现并行化。这种并行称为**数据并行**。SIMD 并行性在大型数据并行问题上非常有用，但是在处理其他并行问题时并不优秀。

SIMD 系统经历了变迁的历史，在 20 世纪 90 年代早期，SIMD 系统的制造者（Thinking Machine 公司）是并行超级计算机最大的制造者。但是到 20 世纪 90 年代末，唯一广泛生产的 SIMD 系统是**向量处理器**。近来，图形处理单元（Graphics Processing Unit，GPU）和台式机的 CPU 利用了 SIMD 计算方面的知识。

#### 向量处理器

30 尽管向量处理器的构成近年来发生了变化，但它们的重要特点还是能够对数组或者数据向量进行操作，而传统的 CPU 是对单独的数据元素或者标量进行操作。近年来，典型的系统有如下特征：

- 向量寄存器。它是能够存储由多个操作数组成的向量，并且能够同时对其内容进行操作的

寄存器。向量的长度由系统决定，从 4 到 128 个 64 位元素不等。

- *向量化和流水化的功能单元*。注意，对向量中的每个元素需要做同样的操作，或者某些操作似于加法的操作，这些操作需要应用到两个向量中相应的元素对上。因此，向量操作是 SIMD。
- *向量指令*。这些是在向量上操作而不是在标量上操作的指令。如果向量的长度是 vector_length，那么这些指令的功能相当于一个简单的循环语句，如，

```
for (i = 0; i < n; i++)
   x[i] += y[i];
```

只需要一次加载、一次加法和一次存储操作就完成了对长度为 vector_length 的数据块的操作，而传统的系统需要对数据块中每个元素单独进行加载、加法和存储操作。

- *交叉存储器*。内存系统由多个内存“体”组成，每个内存体能够独立访问。在访问完一个内存体之后，再次访问它之前需要有一个时间延迟，但如果接下来的内存访问是访问另一个内存体，那么它很快就能访问到。所以，如果向量中的各个元素分布在不同的内存体中，那么在装入/存储连续数据时能够几乎无延迟地访问。
- *步长式存储器访问和硬件散射/聚集*。在步长式存储器访问中，程序能够访问向量中固定间隔的元素，例如能够以跨度 4 访问第一个元素、第五个元素、第九个元素等。（在本文中）散射/聚集是对无规律间隔的数据进行读（聚集）和写（散射）。例如访问第一个元素、第二个元素、第四个元素、第八个元素等。典型的向量系统通过提供特殊的硬件来加速步长式存储器访问和散射/聚集操作。

向量处理器对许多应用都有益处，因为它们速度快而且容易使用。向量编译器擅长于识别向量化的代码。此外，它们能识别出不能向量化的循环而且能提供循环为什么不能向量化的原因。因此，用户能对是否重写代码以支持向量化做出明智的决定。向量系统有很高的内存带宽，每个加载的数据都会使用，不像基于 Cache 的系统不能完全利用高速缓存行中的每个元素。但是，它不能处理不规则的数据结构和其他的并行结构，这对它的**可扩展性**是个限制。可扩展性是指能够处理更大问题的能力。制造一个能够处理长度不断增长的向量的系统是很难的事。新一代系统通过增加向量处理器的数目而不是增加向量长度来进行扩展。当前的商品化系统对短向量提供部分有限的支持，能对长向量进行操作的处理器是定制生产的，非常昂贵。 [31]

**图形处理单元（GPU）**

实时图形应用编程接口使用点、线、三角形来表示物体的表面。它们使用**图形处理流水线**（graphics processing pipeline）将物体表面的内部表示转换为一个像素的数组。这个像素数组能够在计算机屏幕上显示出来。流水线的许多阶段是可编程的。可编程阶段的行为可以通过**着色函数**（shader function）来说明。典型的着色函数一般比较短，通常只有几行 C 代码。因为它们能够应用到图形流中的多种元素（例如顶点）上，所以着色函数一般是隐式并行的。对邻近元素使用着色函数会导致相同的控制流，GPU 可以通过使用 SIMD 并行来优化性能。现在所有的 GPU 都使用 SIMD 并行。这是通过在每个 GPU 处理核中引入大量的 ALU（例如 80 个）来获取的。

处理单个图像就需要大量的数据，数百兆大小的图像是很普通的。因此 GPU 需要维持很高的数据移动速率，另外，为了避免内存访问带来的延迟，GPU 严重依赖硬件多线程。有些系统能够存储数百个挂起线程的状态。实际线程的数目依赖于着色函数需要的资源（例如寄存器）的数量。GPU 的缺点是需要许多处理大量数据的线程来维持 ALU 的忙碌，可能在小问题的处理上性能相对差。

需要强调的是，GPU 不是纯粹的 SIMD 系统。尽管在一个给定核上的 ALU 使用了 SIMD 并行，

但现代的 GPU 有几十个核，每个核都能独立地执行指令流。

GPU 在通用高性能计算中越来越流行，开发出许多语言，使得用户可以利用它们的能力。详见参考文献［30］。

### 2.3.2 MIMD 系统

**多指令多数据流**（Multiple Instruction，Multiple Data，MIMD）系统支持同时多个指令流在多个数据流上操作。因此，MIMD 系统通常包括一组完全独立的处理单元或者核，每个处理单元或者核都有自己的控制单元和 ALU。此外，不同于 SIMD 系统，MIMD 系统通常是**异步的**，即各个处理器能够按它们自己的节奏运行。在许多 MIMD 系统中，没有全局时钟，两个不同处理器上的系统时间之间是没有联系的。实际上，除非程序员强制同步，即使处理器在执行相同顺序的指令
32 时，在任意时刻它们都可能执行不同的语句。

正如我们在第 1 章中提到的，MIMD 系统有两种主要的类型：共享内存系统和分布式内存系统。在**共享内存系统**中，一组自治的处理器通过互连网络（internection network）与内存系统相互连接，每个处理器能够访问每个内存区域。在共享内存系统中，处理器通过访问共享的数据结构来隐式地通信。在**分布式内存系统**中，每个处理器有自己私有的内存空间，处理器 - 内存对之间通过互连网络相互通信。所以在分布式内存系统中，处理器之间是通过发送消息或者使用特殊的函数来访问其他处理器的内存，从而进行显式的通信。见图 2-3 和图 2-4。

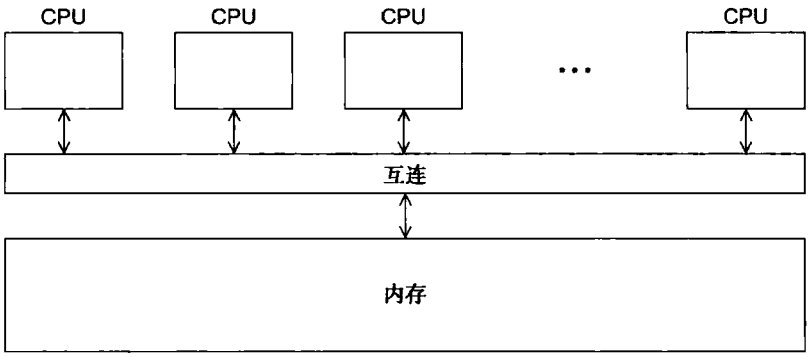



图 2-3　一个共享内存系统

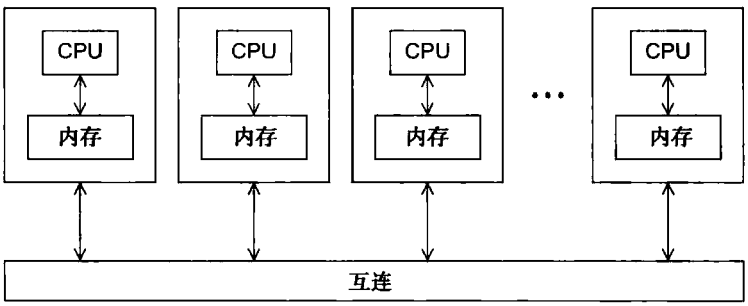



图 2-4　一个分布式内存系统

**共享内存系统**

最广泛使用的共享内存系统使用一个或者多个**多核**处理器。正如第 1 章中讨论的，一个多核处理器在一块芯片上有多个 CPU 或者核。通常，每个核都拥有私有的 L1 Cache，而其他的 Cache
33 可以在核之间共享，也可以不共享。

在拥有多个多核处理器的共享内存系统中，互连网络可以将所有的处理器直接连到主存，或者也可以将每个处理器直接连到一块内存，通过处理器中内置的特殊硬件使得各个处理器可以访问内存中的其他块。如图 2-5 和图 2-6。在第一种系统中，每个核访问内存中任何一个区域的时间都相同；而在第二种系统中，访问与核直接连接的那块内存区域比访问其他内存区域要快很多，因为访问其他内存区域需要通过另一块芯片。因此，第一种系统称为**一致内存访问**（Uniform Memory Access，UMA）系统，而第二种系统称为**非一致内存访问**（Nonuniform Memory Access，NUMA）系统。UMA 系统通常比较容易编程，因为程序员不用担心不同内存区域的不同访存时间。在 NUMA 系统中，对与核直接连接的内存区域的访问速度较快，失去了易于编程的优点，但 NUMA 系统能够比 UMA 系统使用更大容量的内存。

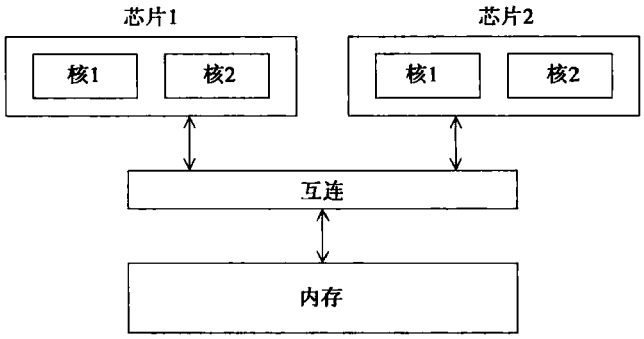



图 2-5　一个 UMA 多核系统

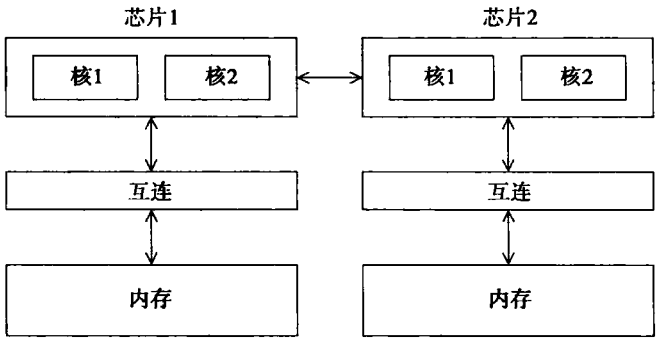



图 2-6　一个 NUMA 多核系统

34

**分布式内存系统**

最广泛使用的分布式内存系统称为**集群**（clusters）。它们由一组商品化系统组成（例如 PC），通过商品化网络连接（例如以太网）。实际上，这些系统中的**节点**（通过通信网络相互连接的独立计算单元），通常都是有一个或者多个多核处理器的共享内存系统。为了将这种系统与纯粹的分布式内存系统分开，这种系统通常称为**混合系统**。现在，通常认为一个集群有多个共享内存节点。

**网格**提供一种基础架构，使地理上分布的计算机大型网络转换成一个分布式内存系统。通常，这样的系统是异构的，即每个节点都是由不同的硬件构造的。

## 2.3.3　互连网络

互连网络（interconnection network）在分布式内存系统和共享内存系统中都扮演了一个决定性的角色，即使处理器和内存无比强大，但一个缓慢的互连网络会严重降低除简单并行程序外所有程序的整体性能。详见习题 2.10。

尽管有些互连网络大体相似，但还是有很多的差别，我们必须对共享内存系统和分布式内存系统的互连网络区别对待。

**共享内存互连网络**

在共享内存系统中，目前最常用的两种互连网络是**总线**（bus）和交叉开关矩阵（crossbar）。

**总线**是由一组并行通信线和控制对总线访问的硬件组成的。总线的核心特征是连接到总线上的设备共享通信线。总线具有低成本和灵活性的优点，多个设备能够以小的额外开销连接到总线上。但是，因为通信线是共享的，因此随着连接到总线设备的增多，争夺总线的概率增大，总线的预期性能会下降。如果将大量的处理器与总线连接，那么可以断定处理器会经常等待访问内存。因此，随着共享内存系统规模的增大，总线会迅速被交换互连网络所取代。

顾名思义，**交换**互连网络使用交换器（switch）来控制相互连接设备之间的数据传递。图2-7a是一个**交叉开关矩阵**（crossbar），线表示双向通信链路，方块表示核或者内存模块，圆圈表示交换器。

单个交换器有两种不同的设置，如图2-7b所示。在使用这种交换器，并且内存模块不比处
35 理器少的情况下，当两个核同时访问相同的内存模块时，它们只可能发生一次冲突。例如，图2-7c显示了，当 $P1$ 向 $M4$ 写数据，$P2$ 从 $M3$ 中读数据，$P3$ 从 $M1$ 中读数据，$P4$ 向 $M2$ 中写数据时，各个交换器的配置情况。

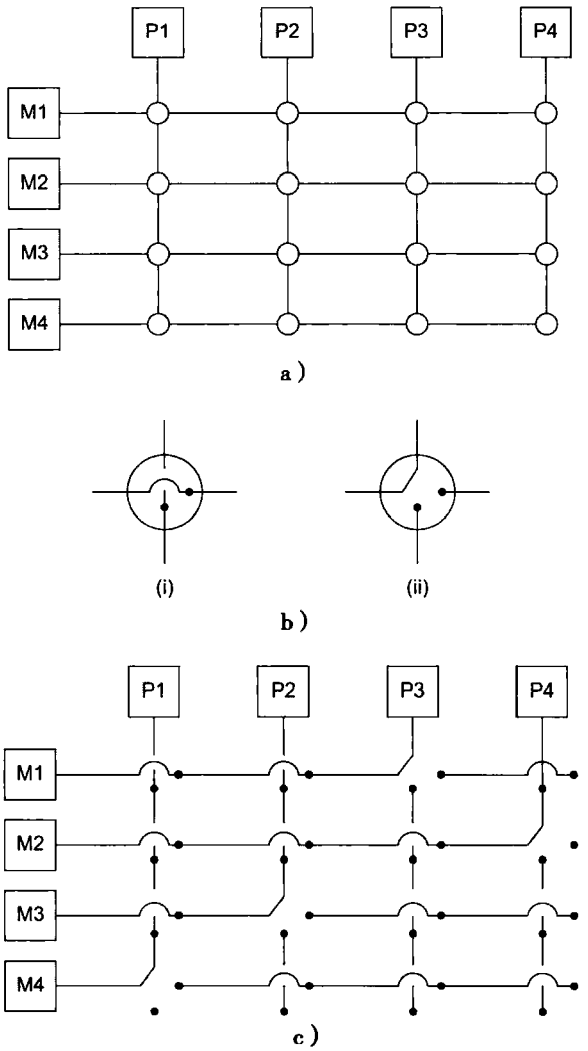



图2-7 a）一个连接4个处理器（$P_i$）和4个内存模块（$M_i$）的交叉开关矩阵；b）交叉开关矩阵内部的交换器；c）多个处理器同时访问内存

交叉开关矩阵允许在不同设备之间同时进行通信，所以比总线速度快。但是，交换器和链路带来的开销也相对高。一个小型的基于总线系统比相等规模的基于交叉开关矩阵系统便宜。

**分布式内存互连网络**

分布式内存互连网络通常分成两种：直接互连与间接互连。在**直接互连**中，每个交换器与一个处理器 - 内存对直接相连，交换器之间也相互连接。图 2-8 给出了一个环（ring）和一个二维**环面网格**（toroidal mesh）。如前面所述，圆圈表示交换器，方块表示处理器，线表示双向通信链路。环比简单的总线高级，因为它允许有多个通信同时发生。然而，在处理器必须等待其他处理器才能完成通信的情况中，制定通信方案会比较容易。环面网格比环昂贵，因为交换器更加复杂。这种交换器需要能够支持 5 个链路而不只是 3 个。如果有 $p$ 个处理器，在环面网格中链路的数目是 $3p$，但在环中只是 $2p$。但是在环面网格中，可以同时通信的链路数目比环中的多，这一点是毋庸置疑的。

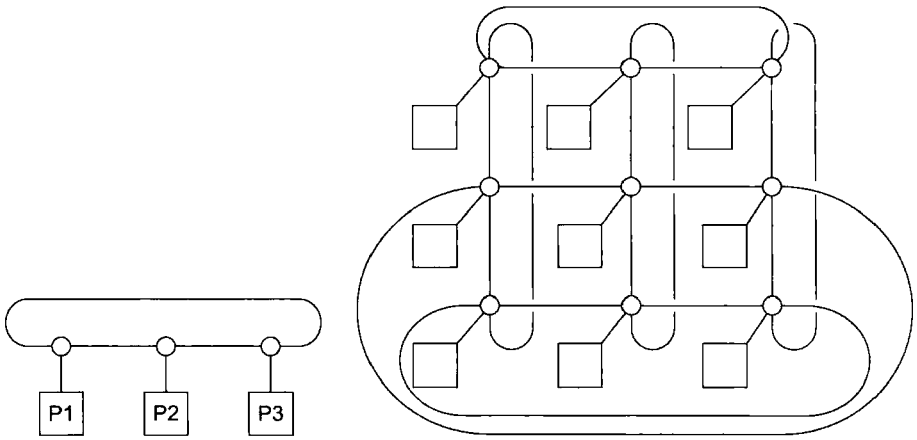



图 2-8　a）一个环；b）一个二维环面网格（toroidal mesh）

衡量“同时通信的链路数目”或者“连接性”的一个标准是**等分宽度**（bisection width）。为了理解这个标准，想象并行系统被分成两部分，每部分都有一半的处理器或者节点。在这两部份之间能同时发生多少通信呢？在图 2-9a 中，我们将一个 8 节点的环分成两组，每组有四个节点，它们之间同时发生通信的次数只为 2（为了使图更容易理解，将每个节点与它的交换器并在一起，随后再直接互连）。但在图 2-9b 中，将节点分成两部分，使它们之间能够同时发生 4 次通信。那么，什么是等分宽度呢？等分宽度是基于最坏情况来估计的，所以等分宽度是 2 而不是 4。

36 / 37

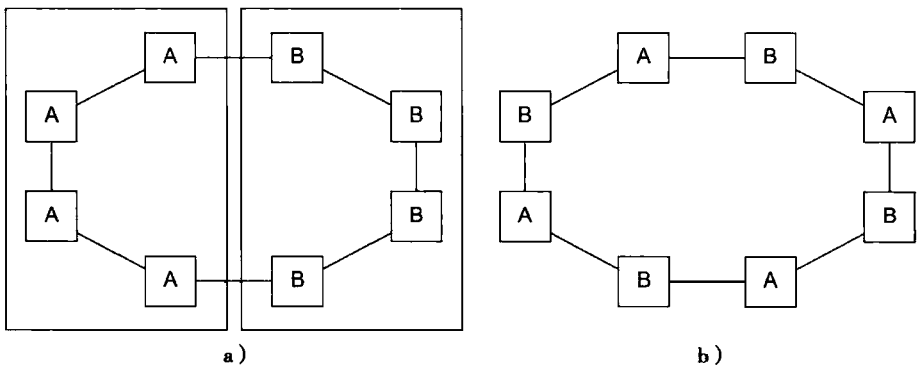



图 2-9　一个环的两种等分：a）在两个等分之间的通信链路数为 2；b）在两个等分之间的通信链路数为 4

计算等分宽度的另一种方法是去除最少的链路数从而将节点分成两等份，去除的链路数就是等分宽度。如果有一个正方形的二维环面网格，有 $p=q^2$ 个节点（$q$ 为偶数），然后通过移除一些“中间”的水平链路和“回绕”的水平链路，将这些节点分成两份。如图 2-10 所示。这意味着等分宽度最多是 $2q=2\sqrt{p}$。实际上，这是最小的可能链路数目，一个正方形二维环面网格的等分宽度就是 $2\sqrt{p}$。

链路的**带宽**（bandwidth）是指它传输数据的速度。通常用兆位每秒或者兆字节每秒来表示。**等分带宽**（bisection bandwidth）通常用来衡量网络的质量。它与等分宽度类似。但是，等分带宽不是计算连接两个等分之间的链路数，而是计算链路的带宽。例如，如果在环中，链路的带宽是 10 亿位每秒，那么环的等分带宽就是 20 亿位每秒或者 2000 兆位每秒。

最理想的直接互连网络是**全相连网络**，即每个交换器与每一个其他的交换器直接连接。如图 2-11所示，它的等分宽度是 $p^2/4$。但是，为节点数目较多的系统构建这样的互连网络是不切实
38 际的，因为它需要总共 $p^2/2+p/2$ 条链路，而且每个交换器都需要连接 $p$ 条链路。因此，这只是一个“理论上可能最佳”的互连网络，它用来作为衡量其他互连网络的基础。

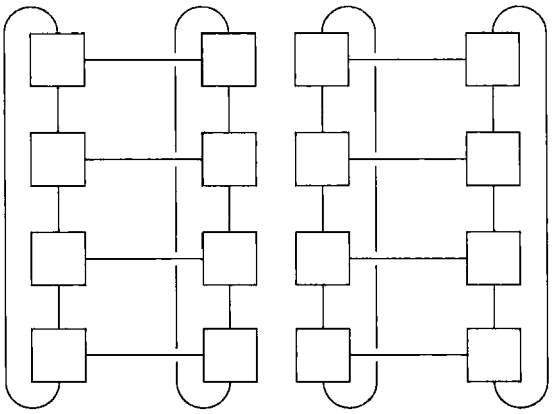


图 2-10　一个二维环面网格的等分

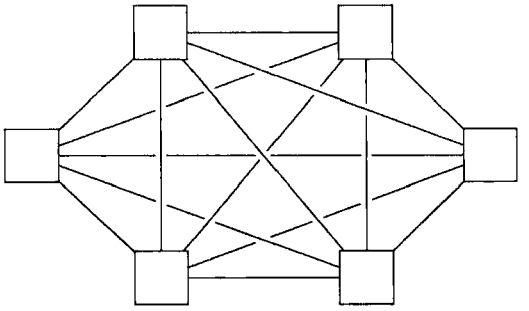
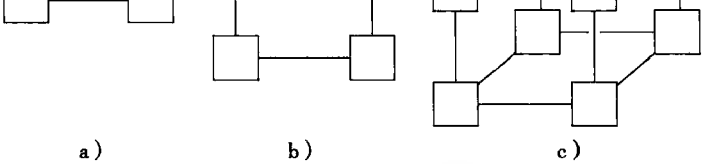


图 2-11　一个全互连网络

**超立方体**是一种已经用于实际系统中的高度互连的直接互连网络。超立方体是递归构造的：一维超立方体是有两个处理器的全互连系统。二维超立方体是由两个一维超立方体组成，并通过“相应”的交换器互连。如图 2-12 所示。因此，维度为 $d$ 的超立方体有 $p=2^d$ 个节点，并且在 $d$ 维超立方体中，每个交换器与一个处理器和 $d$ 个交换器直接连接。这样的超立方体的等分宽度是 $p/2$，所以它比环或者环面网格连接性更高，但需要更强大的交换器，因为每个交换器必须支持 $1+d=1+\log_2(p)$ 条连线，而二维环面网格的交换器只需要 5 条连线。所以构建一个 $p$ 个节点的
39 超立方体互连网络比构建一个二维环面网格更昂贵。



a）　　b）　　c）

图 2-12　a）一维；b）二维；c）三维超立方体

**间接互连**为直接互连提供了一个替代的选择。在间接互连网络中，交换器不一定与处理器直接连接。它们通常由一些单向连接和一组处理器组成，每个处理器有一个输入链路和一个输出链路，这些链路通过一个交换网络连接。见图 2-13。

**交叉开关矩阵**和 **omega 网络**是间接网络中相对简单的例子。前面，我们看到了使用双向链路的共享内存交叉开关矩阵（见图 2-7）。图 2-14 中的交叉开关矩阵通过单向链路共享分布式内存。注意，只要两个处理器不尝试与同一个处理器通信，那么所有的处理器就能够同时与其他的处理器通信。

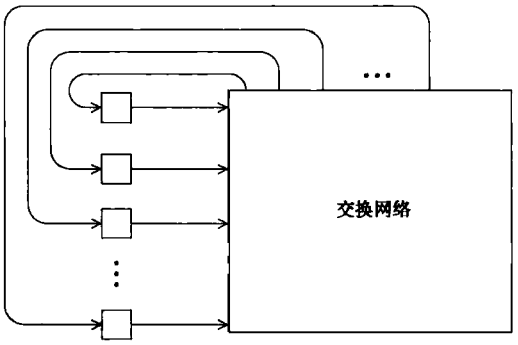



图 2-13　一个通用的间接网络

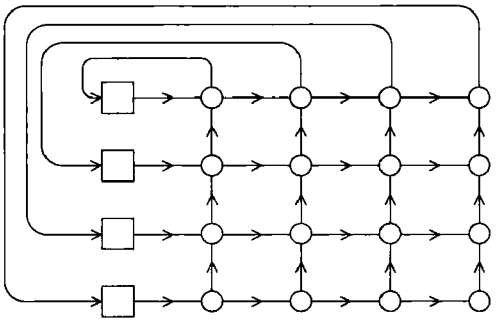


图 2-14　一个用于分布式内存的交叉开关矩阵互连网络

omega 网络见图 2-15。交换器是一个 2 × 2 的交叉开关矩阵（见图 2-16）。注意，与交叉开关矩阵不同的是，有一些通信无法同时进行。例如，在图 2-15 中，处理器 0 给处理器 6 发送一个消息，这时处理器 1 就不能同时给处理器 7 发送消息。另一方面，omega 网络比交叉开关矩阵便宜。omega 网络使用了 $\frac{1}{2}p\log_2(p)$ 个交换器，每个交换器是一个 2 × 2 交叉开关矩阵交换器，所以总共使用了 $2p\log_2(p)$ 个交换器，而交叉开关矩阵使用 $p^2$ 个交换器。

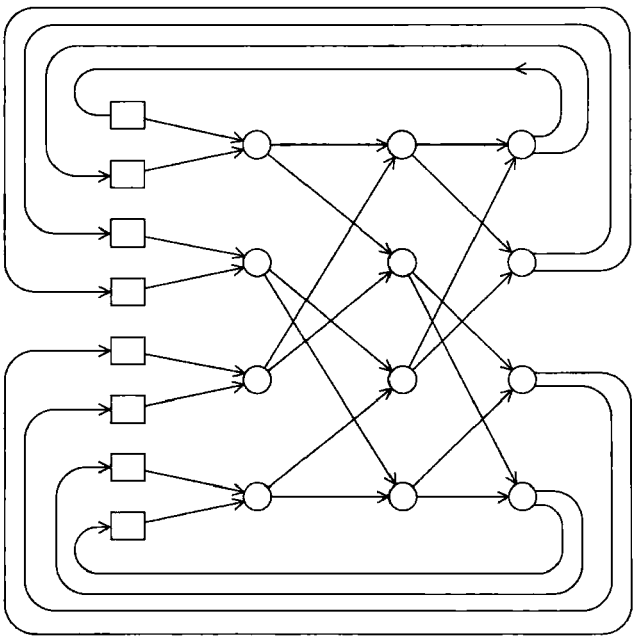


图 2-15　一个 omega 网络

为间接互连网络定义等分宽度就比较复杂，见习题2.14。但原理是相同的：将节点分成大小相等的两部分，并确定在两个等分之间发生多少通信，或者至少需要移除多少数目的链路才能使两个等分不再通信。一个 $p \times p$ 大小的交叉开关矩阵的等分宽度是 $p$，而一个 omega 网络的等分宽度是 $p/2$。

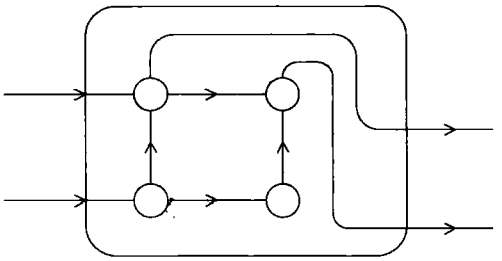


图 2-16 omega 网络中的一个交换器

**延迟和带宽**

当传送数据时，我们关心数据到达目的地需要花多少时间。无论数据是在主存和 Cache 之间传递、在 Cache 和寄存器之间、在磁盘和内存之间，还是在两个分布式内存系统或者混合系统的节点之间传送，这个问题都很关键。下面是两个经常用来衡量互连网络（不管怎么连接）性能的指标：**延迟**（latency）和**带宽**（bandwidth）。延迟是指从发送源开始传送数据到目的地开始接收数据之间的时间。带宽是指目的地在开始接收数据后接收数据的速度。所以如果一个互连网络的延迟是 $l$ 秒，带宽是 $b$ 字节每秒，则传输一个 $n$ 字节的消息需要花费的时间是：

消息传送的时间 $= l + n/b$

但是，需要注意的是，这些术语经常在不同的场合下使用。例如，延迟有时候也用来描述消息传送的总时间。它也用来描述在传送数据时需要的固定开销。例如，如果在分布式内存系统中的两个节点之间传递消息，消息中不仅仅包含原始数据，它可能还包括将要传送的数据、目标地址，有些消息还会描述消息的长度、某些错误校验的信息等。所以，在这种情况下，延迟是指在发送端收集消息的时间、将不同部分组装起来的时间、在接收端将消息拆卸的时间、从消息中抽取原始数据的时间以及在目的地存储的时间的总和。

42

### 2.3.4 Cache 一致性

回忆一下，CPU Cache 是由系统硬件来管理的，程序员对它不能进行直接的控制。这会对共享内存系统带来很多重大的影响。为了理解这些，假如共享内存系统中有两个核，每个核有各自私有的数据 Cache，见图 2-17。如果两个核只对共享数据进行读操作，那么不会发生任何问题。例如，x 是一个共享变量并初始化为 2，y0 是核 0 私有的，y1 和 z1 是核 1 私有的。现在假设按指定时序执行下面的语句：

| 时间 | 核 0 | 核 1 |
|---|---|---|
| 0 | y0 = x; | y1 = 3 × x; |
| 1 | x = 7; | 没有包含 x 的语句 |
| 2 | 没有包含 x 的语句 | z1 = 4 × x; |

那么 y0 的内存区域会最终得到值 2，y1 的内存区域会得到值 6。但是，z1 会获得什么值就不是很清楚了。第一感觉可能是，既然核 0 在 z1 赋值前将 x 更新为 7，所以 z1 可以得到 4 × 7 = 28。但是，在时间 0 时，x 已经在核 1 的 Cache 中。除非出于某些原因，x 清除出核 0 的 Cache，然后再重新加载到核 1 的 Cache 中；若没有上述情况，通常还是会使用原来的值 x = 2，而 z1 也会得到 4 × 2 = 8。

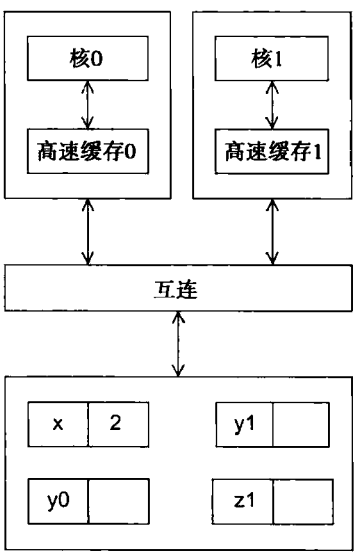



图 2-17 有两个核和两个 Cache 的共享内存系统

43

注意，这种不可预测的行为与系统使用写直达（write-through）策略还是写回（write-back）策略无关。如果使用写直达

策略，主存会通过 x =7 的赋值而更新。但是，这与核 1 中 Cache 的值无关。如果系统使用写回策略，那么当更新 z1 时，x 在核 0 Cache 中的新值对核 1 是不可用的。

显然，这是一个问题。程序员对 Cache 什么时候更新没有直接控制，所以程序不能执行那些看起来无害的、但可能需要访问 z1 值的语句。这里有许多问题，但需要清楚的一点是，单核处理器系统的 Cache 对如下情况没有提供保证：在多核系统中，各个核的 Cache 存储相同变量的副本，当一个处理器更新 Cache 中该变量的副本时，其他处理器应该知道该变量已更新，即其他处理器中 Cache 的副本也应该更新。这称为 Cache 一致性问题。

**监听 Cache 一致性协议**

有两种主要的方法来保证 Cache 的一致性：**监听 Cache 一致性协议**和基于目录的 Cache 一致性协议。监听协议的想法来自于基于总线的系统：当多个核共享总线时，总线上传递的信号都能被连接到总线的所有核“看”到。因此，当核 0 更新它 Cache 中 x 的副本时，如果它也将这个更新信息在总线上广播，并且假如核 1 正在监听总线，那么它会知道 x 已经更新了，并将自己 Cache 中的 x 的副本标记为非法的。这就是监听 Cache 一致性协议大致的工作原理。我们的描述与实际监听协议之间的最大差别在于，广播会通知其他核包含 x 的整个 Cache 行已经更新，而不是只有 x 更新。

关于监听，有几点必须要考虑的。第一，互连网络不一定必须是总线，只要能够支持从每个处理器广播到其他处理器。第二，监听协议能够在写直达和写回 Cache 上都能工作，原则上，如果互连网络可以像总线那样被 Cache 共享，如果是写直达 Cache，那么就不需要额外的互连网络开销，因为每个核都能“监测”写；如果是写回 Cache，那么就需要额外的通信，因为对 Cache 的更新不会立即发送给内存。

**基于目录的 Cache 一致性协议**

不幸的是，在大型网络上，广播是非常昂贵的。监听 Cache 一致性协议每更新一个变量时就需要一次广播（见习题 2.15）。所以监听 Cache 一致性协议是不可扩展的，因为对于大型系统，它会导致性能的下降。例如，假如有一个具有基本分布式内存结构（见图 2-4）的系统，但系统对于所有内存提供单个地址空间。所以，核 0 能够访问核 1 内存中的变量 x，只需要简单地执行一条语句，如 y = x（当然，访问其他核的内存比访问自己的“局部”内存要慢得多，但这是另外一件事）。从原理上说，这样的系统能够扩展到多个核。但是，监听 Cache 一致性协议显然是个问题，因为在互连网络间的广播相对于访问局部内存是相当慢的。 44

**基于目录的 Cache 一致性协议**通过使用一个叫做**目录**（directory）的数据结构来解决上面的问题。目录存储每个内存行的状态。一般地，这个数据结构是分布式的，在我们的例子中，每个核/内存对负责存储一部分的目录。这部分目录标识局部内存对应高速缓存行的状态。因此，当一个高速缓存行被读入时，如核 0 的 Cache，与这个高速缓存行相对应的目录项就会更新，表示核 0 有这个行的副本。当一个变量需要更新时，就会查询目录，并将所有包含该变量高速缓存行置为非法。

显然目录需要大量额外的存储空间，但是，当一个 Cache 变量更新时，只需要与存储这个变量的核交涉（对应的部分目录在这个核上）。

**伪共享**

CPU Cache 是由硬件来实现的，记住这一点非常重要，因为硬件是对高速缓存行进行操作的而不是对单独的变量进行操作。这个特点可能会给性能带来极坏的影响。举例来说，假如需要重复地调用一个函数 f(i,j)，并将计算的值添加到一个向量中：

```
int i, j, m, n;
double y[m];

/* Assign y = 0 */
. . .

for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

为了将程序并行化，我们可以将外部循环的各次迭代分配给各个处理器核来处理。假如有core_count 个核，可能把第一个 m/core_count 个迭代分配给第一个核，下一个 m/core_count 个迭代分配给第二个核，以此类推。

```
/* Private variables */
int i, j, iter_count;

/* Shared variables initialized by one core */
int m, n, core_count
double y[m];

iter_count = m/core_count

/* Core 0 does this */
for (i = 0; i < iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count+1; i < 2*iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

. . .
```

[45]

现在假设共享内存系统有两个核，m = 8，双精度浮点数长度为 8 字节，高速缓存行是 64 字节，y[0]存储在高速缓存行的起始位置。一个高速缓存行能够存储 8 个双精度浮点数，y 占一整个高速缓存行。当核 0 和核 1 同时执行代码时，会发生什么情况呢？因为 y 的所有值都存储在单个高速缓存行中，所以每次一个核执行语句 y[i] += f(i,j)时，高速缓存行就会失效，当另一个核尝试执行该语句时，就必须从内存中将更新过的高速缓存行取出。所以，如果 $n$ 很大，尽管核 0 和核 1 不会访问 y 中对方的元素，但我们能预料到，大量的赋值操作 y[i] += f(i,j)会访问主存。这称为**伪共享**。因为系统表现的好像核之间会共享 y。

注意，伪共享不会引发错误结果，但是，它能引起过多不必要的访存，降低程序的性能。可以通过在线程或者进程中临时存储数据，再把临时存储的数据更新到共享存储来降低伪共享带来的影响。第 4 章和第 5 章将探讨这个问题。

### 2.3.5 共享内存与分布式内存

并行计算的新手有时想知道为什么不是所有的 MIMD 系统都是共享内存的，因为大多数程序员觉得通过共享数据结构隐式地协调多个处理器的工作，比显式地发送消息更吸引人。但这里有一些问题，我们会在讨论分布式或者共享内存软件时提到一些。但是，主要的硬件方面的问题是互连网络扩展的代价。当向总线增加处理器时，访问总线发生冲突的可能性骤升。所以总线适合于那些处理器数目较少的系统。大型的交叉开关矩阵是非常昂贵的，所以使用大型交叉开关矩阵互连的系统也是比较少见的。另一方面，分布式内存互连网络，如超立方体、环面网格相对便宜一些，有成千上万个处理器的分布式系统就是用这种互连网络或者其他构建的。因此，分布式内存系统比较适合于那些需要大量数据和计算的问题。

[46]

## 2.4　并行软件

并行硬件的时代已经来了。几乎所有的台式机和服务器都使用多核。但对并行软件不适用。除了操作系统、数据库系统、Web 服务器外，目前能够充分利用并行硬件特点的商业软件很少。正如第 1 章所提到的，不再能通过硬件和编译器为应用提供性能上的稳定增长了。假如我们继续追求应用性能和应用功效上的增长，软件开发人员必须学会编写能够利用共享内存或者分布式内存体系结构潜力的应用程序。本节中，我们简要地学习编写并行系统上的软件所涉及的问题。

首先是一些术语。通常，在运行共享内存系统时，会启动一个单独的进程，然后派生（folk）出多个线程。所以当我们谈论共享内存程序时，我们指的是正在执行任务的线程。另一方面，当我们运行分布式内存程序时，我们使用的是多个处理器，所以我们指的是正在执行任务的进程。当讨论对共享内存系统和分布式内存系统同样适用时，我们指的是执行任务的进程或者线程。

### 2.4.1　注意事项

在继续介绍之前，需要强调本节的一些限制。首先，在本书余下的部分中，我们只讨论 MIMD 系统的软件。例如，尽管使用 GPU 作为并行计算的平台在迅速增多，但 GPU 的应用程序编程接口与标准的 MIMD 的 API 有很大差别。其次，我们强调，我们所涉及的只是给出问题的概念而不是尝试深入理解问题。

最后，我们主要关注的是称为**单程序多数据流**（Single Program，Multiple Data，SPMD）程序。SPMD 程序不是在每个核上运行不同的程序，相反，SPMD 程序仅包含一段可执行代码，通过使用条件转移语句，可以让这一段代码在执行时表现得像是在不同处理器上执行不同的程序。例如，

```
if (I'm thread/process 0)
   do this;
else
   do that;
```

可以看到，SPMD 程序也能够实现数据并行，例如，

```
if (I'm thread/process 0)
   operate on the first half of the array;
else /* I'm thread/process 1 */
   operate on the second half of the array;
```

如果一个程序是通过将任务划分，分给各个进程或者线程来实现并行，则称它是**任务并行**（task parallel）。第一个例子清楚地表明，SPMD 程序也能够实现**任务并行性**（task parallelism）。 [47]

### 2.4.2　进程或线程的协调

只有在极少数的情况下，获取好的并行性能是容易的。例如，假如有两个数组，我们想要将它们相加：

```
double x[n], y[n];
. . .
for (int i = 0; i < n; i++)
   x[i] += y[i];
```

为了并行化这段代码，只需要将数组中的元素分配给线程或者进程。例如，假如有 $p$ 个进程/线程，我们可能让进程/线程 0（也可称为 0 号进程/线程）负责元素 0，…，$n/p-1$ 的相加，进程/线程 1 负责 $n/p$，…，$2n/p-1$ 元素的相加，以此类推。

所以，在这个例子中，程序员只需要做：

（1）将任务在进程/线程之间分配

1）这个分配可以使得每个进程/线程获得大致相等的工作量，并且

2）这个分配可以使得需要的通信量是最小的。

需要在进程/线程之间平均分配任务从而满足条件1），这称为**负载均衡**（load balancing）。分配任务需要满足的这两个条件是显而易见的，但也是非常重要的。在许多情况下，不需要对它们进行过多的思考，它们通常在程序员不事先知道工作量而是在程序运行时生成工作量的情况下才需要考虑。例如，第6章中的树搜索问题。

将串行程序或者算法转换为并行程序的过程称为**并行化**（parallelization）。某些程序，如果能够通过简单地将任务分配给进程/线程来实现并行化，我们称该程序是**易并行的**（embarrassingly parallel）。不幸的是，程序员编写易并行的程序并不容易，相反，如果能成功设计出一种将任何问题都并行化的方法，那么这真是一件大喜事。

但是，大部分问题是很难找到并行方案的。正如在第1章中看到的，对于这些问题，我们需要协调进程/线程之间的工作。在这些程序中，通常还需要：

（2）安排进程/线程之间的同步

（3）安排进程/线程之间的通信

最后两个问题往往是相关的。例如，在分布式内存程序中，经常通过进程间通信来隐式地同
[48] 步进程；而在共享内存系统中，经常需要通过同步来实现线程间的通信。我们会在下面详细地探讨这两个问题。

### 2.4.3 共享内存

正如我们在前面提到的，在共享内存系统中，变量可以是**共享的**（shared）或者**私有的**（private）。共享变量可以被任何线程读、写，而私有变量只能被单个线程访问。线程间的通信是通过共享变量实现的，所以通信是隐式的，而不是显式的。

**动态线程和静态线程**

在许多情况下，共享内存程序使用的是**动态线程**。在这种范式中，有一个主线程，并在任何时刻都有一组工作线程（可能为空）。主线程通常等待工作请求（例如，通过网络），当一个请求到达时，它派生出一个工作线程来执行该请求。当工作线程完成任务，就会终止执行再合并到主线程中。这种模式充分利用了系统的资源，因为线程需要的资源只在线程实际运行时使用。

另一种程序运行模式是**静态线程**范式。在这种范式中，主线程在完成必需的设置后，派生出所有的线程，在工作结束前所有的线程都在运行。当所有的线程都合并到主线程后，主线程需要做一些清理工作（如释放内存），然后也终止。在资源利用方面，这个范式可能不是很高效：如果线程空闲，它的资源（如栈、程序计数器等）不能被释放。但是，线程的派生和合并操作是很耗时的。所以如果所需的资源是可用的，静态线程模式有潜力比动态线程获得更高的性能。它也更加接近于分布式内存编程中最广泛使用的模式。既然静态线程范式适用于一种系统，也适用于另一种系统，因此，我们会经常使用静态线程范式。

**非确定性**

在任何一个MIMD系统中，如果处理器异步执行，那么很可能会引发**非确定性**。给定的输入能产生不同的输出，这种计算称为非确定性。如果多个线程独立执行任务，每次运行时它们完成语句的速度各不相同，那么程序的结果也不同。举个简单的例子，有两个线程，一个标志符为0，另一个为1，每个线程都有一个私有变量my_x。线程0中my_x的值是7，而线程1中my_x的值
[49] 是19。此外，假设两个线程都执行下面的代码：

```
. . .
printf("Thread %d > my_val = %d\n", my_rank, my_x);
. . .
```

输出可能是：

```
Thread 0 > my_val = 7
Thread 1 > my_val = 19
```

但也有可能是：

```
Thread 1 > my_val = 19
Thread 0 > my_val = 7
```

实际上，情况可能更糟糕，一个线程的输出可能被另一个线程的输出所打断。然而，这里的情况是，因为线程独立执行并且独立地与操作系统交互，执行一个线程完成一段语句所花的时间在不同次的执行也是不同，所以语句执行的顺序是不能预测的。

在许多情况下，非确定性并不是问题。在我们的例子中，因为我们将输出标记了线程的标号，所以输出的顺序就没有关系了。但在其他情况下，尤其是在共享内存程序中，非确定性是灾难性的，因为它们很容易导致程序错误。下面是一个简单的例子。

假设每个线程计算一个 int 型整数，这个 int 型整数存储在私有变量 my_val 中。假设想要将 my_val 的值加到共享内存的 x 位置中，x 初始化为 0。两个线程都要执行下面的代码：

```
my_val = Compute_val(my_rank);
x += my_val;
```

一次加法操作通常需要将两个数加载到寄存器、相加，最后存储结果。为了使过程简单化，假设值加载时，是从主存中直接加载到寄存器中；在存储时，直接将值从寄存器存储到主存中。下面是一个可能的事件顺序：

| 时间 | 核 0 | 核 1 |
|---|---|---|
| 0 | 完成对 my_val 的赋值 | 正在调用 Compute_val |
| 1 | 将 x = 0 装载入寄存器 | 完成对 my_val 的赋值 |
| 2 | 将 my_val = 7 装载入寄存器 | 将 x = 0 装载入寄存器 |
| 3 | 将 my_val = 7 与 x 相加 | 将 my_val = 19 装载入寄存器 |
| 4 | 存储 x = 7 | 将 my_val 与 x 相加 |
| 5 | 开始其他工作 | 存储 x = 19 |

显然这不是我们想要的，也很容易想象到，有其他事件顺序会产生一个 x 的不正确值。这里的非确定性是两个线程尝试同时更新内存区域 x 而造成的。当线程或者进程尝试同时访问一个资源时，这种访问会引发错误，我们经常说程序有**竞争条件**（race condition），因为线程或者进程处 50
于竞争状态下。即程序的输出依赖于赢得竞争的进程或者线程。在我们的例子中，线程竞争执行 x + = my_val。在这种情况下，除非一个线程在另一个线程开始前，计算完成 x + = my_val，结果才是正确的。一次只能被一个线程执行的代码块称为**临界区**（critical section），通常是程序员的责任来保证**互斥地**访问临界区。换句话说，我们需要保证如果一个线程在临界区中执行代码，其他线程需要被排除在临界区外。

保证互斥执行的最常用机制是**互斥锁**（mutual exclusion clock），或者**互斥量**（mutex），或者锁（lock）。互斥量是由硬件支持的一个特殊类型的对象。基本思想是每个临界区由一个锁来保护。在一个线程能够执行临界区中的代码前，它必须通过调用一个互斥量函数来获取互斥量，在执行完临界区代码时，通过调用解锁函数来释放互斥量。当一个线程“拥有”锁时，即从调用加锁函数返回但还没有调用解锁函数时，其他线程尝试执行临界区中的代码必须在调用加锁函数时

等待。

因此，为了保证代码正常运行，我们必须修改代码，使得它看起来像：

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

这保证了每次只有一个线程执行语句 x + =my_val。这段代码在线程上没有施加任何预定的顺序。或者线程 0 或者线程 1 能够先执行 x + =my_val。

还需要注意的是，使用互斥量加强了临界区的**串行性**（serialization）。因为在临界区中，一次只有一个线程能执行代码。代码被有效地串行化了。因此，我们希望代码尽可能少地包含临界区，并且临界区尽可能地短。

还有其他可以替代互斥量的方式。在**忙等待**（busy-waiting）时，一个线程进入一个循环，这个循环的目的只是测试一个条件。在我们的例子中，假设共享变量 ok_for_1 初始化为 false，下面的代码能够保证只有在线程 0 将 ok_for_1 置为 ture 后，线程 1 才能更新 x：

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
   while (!ok_for_1);  /* Busy-wait loop */
x += my_val;           /* Critical section */
if (my_rank == 0)
   ok_for_1 = true;    /* Let thread 1 update x */
```

所以，直到线程 0 执行 ok_for_1 =true 完后，线程 1 一直陷在循环 while(! ok_for_1)中，
51 这个循环称为“忙等待”，因为线程忙着等待条件。这个程序的优点是易于理解和实现。但是，它浪费系统的资源，因为即使线程在做无用功，执行该线程的核还是会重复的检查是否能进入临界区。**信号量**（semaphore）与互斥量类似，尽管它们的行为细节有些许不同。对某些类型的线程，使用信号量实现同步比用互斥量实现要简单。**监视器**（monitor）能够在更高层次提供互斥执行。监视器是一个对象，这个对象的方法，一次只能被一个线程执行。我们将在第 4 章中讨论繁忙等待和信号量。

目前，还有很多其他同步方法正在研究中，但没有被广泛地使用。其中受到最多关注的是**事务内存**（transactional memory）。在数据库管理系统中，**事务**是系统访问数据库的单位。例如，从你的储蓄账户向你的支票账户转账 1000 美元，银行软件应该认为是一个事务，所以软件在没有将钱款加入你的支票账户时，不会减少你储蓄账户中的金额。如果软件减少了你储蓄账户中的金额，却没有将钱款加入你的支票账户，事务就要回滚。换句话说，事务应该要么全部执行，要么都不执行。事务内存背后的基本思想是将共享内存系统中的临界区看做事务。要么一个线程成功地完成临界区代码，要么所有的部分结果回滚，临界区代码重复执行。

**线程安全性**

在许多、但不是大部分情况下，并行程序能够调用为串行程序开发的函数，并且不会产生问题。但是，有一些值得注意的例外。对于 C 程序员来说，最重要的例外是使用静态局部变量的函数。普通 C 语言局部变量（在函数中声明的变量），是从系统栈中分配出来的。因为每个线程有自己的栈，所以普通的 C 局部变量是私有的。但是，在函数中声明的静态变量，在函数调用时不会被销毁。因此，静态变量能够被调用函数的线程共享，这会引起无法预测和不必要的后果。

例如，C 语言 String 库函数的 strtok，将一个输入字符串分成多个子字符串。当它第一次调用时，传递一个待分割的字符串，而在随后的调用中，它返回分割好的连续的子串。在第一次调用时，通过一个静态的 char * 变量来指示传递给它的字符串。现在，假设有两个线程要将字符串分割成字串。首先，线程 0 对 strtok 进行了首次调用，然后在线程 0 完成分割字串操作完

成前，线程 1 就进行了对 strtok 的首次调用，这样会导致线程 0 的字符串丢失或者覆盖，在随后的调用时它会得到线程 1 的字串。

类似 strtok 这样的函数不是**线程安全的**。这意味着，如果它被多线程程序使用，那么会产生错误或者未知结果。当一段代码不是线程安全的，通常是因为不同的线程在访问共享的数据。[52] 因此，正如我们看到的，即使许多串行程序能够在多线程程序中安全地使用（即它们是线程安全的），程序员仍然需要谨慎使用那些专门为串行程序编写的函数。我们将在第 4 章和第 5 章中研究线程安全。

### 2.4.4　分布式内存

在分布式内存程序中，各个核能够直接访问自己的私有内存。目前已经有了许多可以使用的分布式内存编程 API。但是，最广泛使用的是消息传递。所以，在本节中，我们主要介绍消息传递。然后，我们会简略地看看其他的、较少使用的 API。

也许对于分布式内存应用程序编程接口来说，首先要考虑的一点就是：它也能在共享内存硬件上使用。这完全是可行的。对于程序员来说，只是从逻辑上将共享内存分割成私有的地址空间，给各个线程使用，并使用库函数或者编译器实现所需要的通信。

正如我们前面提到的，分布式内存程序通常执行多个进程而不是多个线程。这是因为在分布式内存系统中，典型的“执行的线程”是在独立的 CPU 中独立的操作系统上运行的，目前还没有软件架构可以启动一个简单的“分布式”进程，使该进程能在系统中的各个节点上再派生出更多的线程来。

**消息传递**

消息传递的 API（至少）要提供一个发送和一个接收函数。进程之间通过它们的序号（rank）互相识别。序号的范围从 0 ~ $p-1$，其中 $p$ 表示进程的个数。例如，进程 1（也可称为 1 号进程）可以使用下面的伪代码向进程 0 发送消息：

```
char message[100];
. . .
my_rank = Get_rank();
if (my_rank == 1) {
   sprintf(message, "Greetings from process 1");
   Send(message, MSG_CHAR, 100, 0);
} else if (my_rank == 0) {
   Receive(message, MSG_CHAR, 100, 1);
   printf("Process 0 > Received: %s\n", message);
}
```

这里的 Get_rank 函数返回调用进程的序号。然后进程的分支依赖于它们的序号。进程 1 用 C 标准库中的 sprintf 函数创建消息，并且调用 Send 发送消息给进程 0。函数调用的参数依次是：消息、消息元素的类型（MSG_CHAR）、消息中元素的个数（100）、目标进程的序号（0）。另一方面，进程 0 使用下列参数调用 Receive 函数：存放将要接收到消息的变量（message）、消息元素的类型、能够存储消息元素的个数和发送消息进程的序号。在完成调用 Receive 后，进程 0 [53] 将消息打印出来。

这里有几点值得注意。第一点，程序段是 SPMD。两个进程使用相同的可执行代码，但执行不同的操作。在这种情况下，它们所执行的操作依赖于它们的序号。第二点，在不同进程中，变量 message 指的是不同的内存块。程序员经常通过使用如 my_message、local_message 这样的变量名来强调这一点。第三点，我们假设线程 0 能够写 stdout。通常情况下，即使消息传递 API 没有显式的支持，但大多数实现消息传递的 API 程序都允许所有的进程访问 stdout 和 stderr。我们将在后面进一步探讨 I/O。

Send 和 Receive 函数的行为可以有很多种，大多数消息传递 API 提供多个不同的发送和接收函数。调用 Send 函数最简单的行为是**阻塞**（block）直到对应的 Receive 函数开始接收数据为止。这意味着 Send 函数调用不会返回，直到对应的 Receive 函数启动为止。另一种选择可以是，Send 函数将消息的内容复制到它私有的存储空间中，在数据复制完之后立即返回。Receive 函数最常见的行为是阻塞直到消息被接收。Send 和 Receive 函数还有其他可能的实现方式，我们将在第 3 章中讨论。

典型的消息传递 API 还提供了许多其他的函数。例如，提供各种“集合”（collective）通信的函数，如**广播**（broadcast）。在广播通信中，单个进程传送相同的数据给所有的进程。又例如**归约**（reduction），在归约函数中，将各个进程计算的结果汇总成一个结果。例如，对各个进程计算出的值相加求总和。还有一些管理进程和复杂数据结构通信的特殊函数。对于消息传递，最常使用的 API 是**消息传递接口**（Message Passing Interface，MPI）。我们将在第 3 章中深入研究。

消息传递是开发并行程序的利器。几乎世界上所有运行在最强计算机上的程序都使用了消息传递。但是，它是非常底层的，程序员需要管理很多细节。例如，为了将一个串行程序并行化，通常需要重写大部分程序。程序中的数据结构要么被每个进程复制，要么被显式地分布在各个进程中。此外，重写程序不能增量地完成。如果一个数据结构在程序中的多个部分使用，在并行部分将它分布在各个进程之中，在串行部分将它收集，这样的代价会比较昂贵。因此，消息传递有
54 时称为“并行编程的汇编语言”，因此也有许多尝试开发其他分布式内存 API 的方法。

**单向通信**

在消息传递中，一个进程必须调用一个发送函数，并且发送函数必须与另一个进程调用的接收函数相匹配。任何通信都需要两个进程的显式参与。在**单向通信**（one-sided communication）或者**远程内存访问**（remote memory access）中，单个处理器调用一个函数。在这个函数中，或者用来自另一个进程的值来更新局部内存，或者使用来自于调用进程的值更新远端内存。这种方式能够简化通信，因为它只需要一个进程的参与。此外，它还消除了两个进程间同步的代价，有效地降低了通信的开销。它取消了一个函数（发送或者接收），也可以减少开销。

但是，实际上其中的某些优点是很难实现的。例如，如果进程 0 将一个值复制到进程 1 的内存空间中，那么进程 0 必须有一些方法来知道复制的安全性。因为它可能会将某些内存区域覆盖。进程 1 也必须有一些方法来知道什么时候内存区域被更新了。第一个问题可以通过在复制前，将两个进程同步来解决。第二个问题可以通过另一次同步或者使用一个“标志”变量来解决，这个标志变量是在进程 0 完成复制时设立的。在后一种方法中，进程 1 需要**轮询**（poll）标志变量，直到它得到一个表示复制的数据已经可用的值。进程 1 必须不断的检查标志变量的值，直到得到一个值表明进程 0 已经完成复制。显然，这些问题会大大地增加传送数据的开销。更大的困难是，因为两个进程间没有显式的交互，所以远程内存操作会引起错误并且很难被追踪。

**划分全局地址空间的语言**

许多编程人员发现，共享内存编程比消息传递或者单向通信更加吸引人。有些研究小组正在开发允许在分布式内存硬件上使用共享内存技术的并行编程语言。这可不像听起来那么简单。如果只是简单地编写一个编译器，这个编译器将分布式系统中所有分散的内存看做一个大内存，那么程序性能往往会比较差，即使在最好的情况下，程序性能也是不可预测的。因为每次一个执行进程访存，它访问的可能是局部内存，即只属于当前执行核的内存，也可能是远端内存，即属于其他核的内存。访问远端内存的时间是访问局部内存时间的数百倍甚至数千倍。举个例子，考虑下面的共享内存向量加法的伪代码：

```
shared int n = . . . ;
shared double x[n], y[n];
private int i, my_first_element, my_last_element;
my_first_element = . . . ;
my_last_element = . . . ;
/* Initialize x and y */
. . .

for (i = my_first_element; i <= my_last_element; i++)
   x[i] += y[i];
```

首先声明两个共享数组 x 和 y。然后在进程序号的基础上，我们决定哪个元素“属于”哪个进程。在数组初始化之后，每个进程将它们各自分配到的 x 和 y 数组中对应的元素相加求和。如果 x、y 中分配给每个进程的元素都正好存储在运行该进程的核所拥有的内存中，那么执行代码的速度会非常快。但是，如果所有 x 数组的元素都分配给核 0，所有 y 数组的元素都分配给给核 1，那么程序的性能会非常糟糕，因为每次执行赋值操作 x[i] += y[i]时，进程都需要访问远程内存。

**划分全局地址空间**（Partitioned Global Address Space，PGAS）语言提供了一些共享内存程序的机制。它们给程序员提供了一些工具，避免上面讨论的问题发生。私有变量在运行程序的核的局部内存空间中分配，共享数据结构中数据的分配由程序员控制。所以，程序员知道共享数组中哪个元素是在进程的本地内存中。

有很多研究项目在研究 PGAS 语言的开发，如 [7，9，45]。

### 2.4.5　混合系统编程

我们应该注意到，在类似多核处理器集群的系统中使用混合编程方式，即在节点上使用共享内存 API，而在节点间通信使用分布式内存 API，是完全可能的。但是，通常这只应用于那些想获得高性能的系统，因为混合系统 API 的复杂性使得程序开发极其困难。可以参考文献 [40] 中的例子。一般情况下，这样的系统通常用分布式内存 API 来实现节点内和节点间的通信。

## 2.5　输入和输出

通常我们尽量避免输入和输出方面的问题。这里有很多原因。首先也是最重要的是，并行输入输出、多个核访问多个磁盘或者其他设备，可以专门用一本书来阐述，如参考文献 [35]。其次，我们开发的大部分程序仅有很少的输入和输出。它们读、写的数据量非常小，很容易通过标准 C 函数 printf、fprintf、scanf 和 fscanf 来管理。但是，尽管我们很少使用这些函数，但它们也可能引起一些问题。因为这些函数是标准 C 函数的一部分，而 C 是串行语言，并没有考虑到这些函数被不同的进程调用时，会发生什么。另一方面，单个进程派生出的多个线程共享 stdin、stdout 和 stderr。但是（正如我们看到的），当多个线程尝试访问 stdin、stdout 和 stderr 其中的一个时，结果是非确定的，不能预测会发生什么。

55 / 56

当从多个进程调用 printf 函数时，作为开发人员，我们想要结果输出在某一个单一系统的显示屏上，这个系统是我们启动程序的那台机器。实际上，大部分系统都是这么做的。但是我们的期望并不能得到保证，系统可能会做一些其他的。例如，只有一个进程有权访问 stdout 或 stderr，甚至没有进程有权访问 stdout 或 stderr。

当我们运行多个进程时，调用 scanf 函数会发生什么还不是很明确。输入应该在各个进程间平分吗？或者只有一个进程能够调用 scanf 吗？大部分系统允许至少一个进程调用 scanf，通常是进程 0，而有些系统允许更多。再强调一遍，有些系统不允许任何进程调用 scanf 函数。

当多个进程能够访问 stdout、stderr 或者 stdin 时，如你所猜想的，输入的分布和输出

的顺序是非确定的。对于输出，数据可能在每一次程序运行时以不同的顺序出现，或者更糟糕的情况，一个进程的输出被另一个进程的输出打断。对于输入，即使每一次输入的内容都一样，在多次运行时，每个进程读到的数据是不同的。

为了能够部分地解决这些问题，当并行程序需要输入/输出时，我们会做一些假设并遵循一些规则：

- 在分布式内存程序中，只有进程 0 能够访问 stdin。在共享内存程序中，只有主线程或者线程 0 能够访问 stdin。
- 在分布式内存和共享内存系统中，所有进程/线程都能够访问 stdout 和 stderr。
- 但是，因为输出到 stdout 的非确定性顺序，大多数情况下，只有一个进程/线程会将结果输出到 stdout。但输出调试程序的结果是个例外，在这种情况下，允许多个进程/线程写 stdout。
- 只有一个进程/线程会尝试访问一个除 stdin、stdout 或者 stderr 外的文件。所以，例如，每个进程/线程能够打开自己私有的文件进行读、写，但是没有两个进程/线程能打开相同的文件。

57

- 调试程序输出在生成输出结果时，应该包括进程/线程的序号或者进程标识符。

## 2.6 性能

编写并行程序的主要目的当然是提高性能，那么我们应该期望什么？应该怎么样评价程序？

### 2.6.1 加速比和效率

通常，我们最希望达到的是，任务在核之间平均分配，又不会为每个核引入额外的工作量。如果我们能成功达到目标，当在 $p$ 核系统上运行程序，每个核运行一个进程或者线程，并行程序的运行速度就是串行程序速度的 $p$ 倍。如果我们称串行运行时间为 $T_{串行}$，并行运行时间为 $T_{并行}$，那么最佳的预期是 $T_{并行} = T_{串行}/p$。此时，我们称并行程序有**线性加速比**（linear speedup）。

实际上，我们不可能获得线性加速比，因为多个进程/线程总是会引入一些代价。例如，共享内存程序通常都有临界区，需要使用一些互斥机制，如互斥量。调用互斥量是串行程序没有的代价，使用互斥量就会强制并行程序串行执行临界区代码。而分布式内存程序通常需要跨网络传输数据，这比访问局部内存中的数据慢。相反，串行程序没有这些额外的开销。因此，找到一个具有线性加速比的并行程序是非常不容易的。此外，随着进程/线程个数的增多，开销也会增大。更多的线程意味着更多的线程需要访问临界区，更多的进程意味着更多的数据需要跨网络传输。

所以，我们定义并行程序的**加速比**（speedup）是：

$$S = \frac{T_{并行}}{T_{串行}}$$

线性加速比为 $S = p$，这是非常难以达到的。此外，随着 $p$ 的增加，希望 $S$ 越来越接近理想的线性加速比 $p$。这里可以换一种说法来理解：$S/p$ 随着 $p$ 的增大越来越小。表 2-4 给出了一个随着 $p$ 的变化，$S$ 和 $S/p$ 变化的例子。$S/p$ 的值[⊖]，有时也称为并行程序的**效率**。如果替换公式中的 $S$，可以看到效率可以表示为：

58

$$E = \frac{S}{p} = \frac{\dfrac{T_{串行}}{T_{并行}}}{p} = \frac{T_{串行}}{p \cdot T_{并行}}$$

⊖ 这些数据来自第 3 章，详见表 3-6 和表 3-7。

表 2-4　一个并行程序的加速比和效率

| $p$ | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| $S$ | 1.0 | 1.9 | 3.6 | 6.5 | 10.8 |
| $E = S/p$ | 1.0 | 0.95 | 0.90 | 0.81 | 0.68 |

表 2-5　不同问题规模的一个并行程序的加速比和效率

| | $p$ | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|
| 一半 | $S$ | 1.0 | 1.9 | 3.1 | 4.8 | 6.2 |
| | $E$ | 1.0 | 0.95 | 0.78 | 0.60 | 0.39 |
| 原始 | $S$ | 1.0 | 1.9 | 3.6 | 6.5 | 10.8 |
| | $E$ | 1.0 | 0.95 | 0.90 | 0.81 | 0.68 |
| 双倍 | $S$ | 1.0 | 1.9 | 3.9 | 7.5 | 14.2 |
| | $E$ | 1.0 | 0.95 | 0.98 | 0.94 | 0.89 |

显然 $T_{并行}$、$S$、$E$ 依赖于 $p$，即进程或者线程的数目。还需要记住，$T_{并行}$、$S$、$E$ 和 $T_{串行}$还依赖于问题的规模。例如，如果将表 2-4 中的问题规模加倍，可以得到表 2-5 中的加速比和效率，这个加速比见图 2-18，效率见图 2-19。

在这个例子中，我们可以看到当问题的规模变大时，加速比和效率增加；当问题的规模变小时，加速比和效率降低。这是正常的。许多并行程序将串行程序的任务分割开来，在进程/线程之间分配，并增加了必需的“并行开销”，如互斥或通信。因此，如果用 $T_{开销}$表示并行开销，那么

$$T_{并行} = T_{串行}/p + T_{开销}$$

此外，随着问题规模的增加，$T_{开销}$比 $T_{串行}$增长得慢。参见习题 2.16。直觉会告诉你，进程/线程有更多的任务去做，用于协调进程/线程的工作所需要的相对时间变少了。

最后需要考虑的问题是，在计算加速比和效率时，$T_{串行}$应该使用什么值。有些作者认为 $T_{串行}$应该是在最强的核上运行串行程序的最快时间。而实际上，许多作者将串行程序在并行系统上单个核的运行时间作为 $T_{串行}$。所以，当研究并行 shell 排序程序的性能时，第一组作者会在最快系统 59
的一个核上运行串行基数排序或者快速排序，而第二组作者会在并行系统的一个核上运行串行的 shell 排序。通常情况下，我们使用第二种方法。

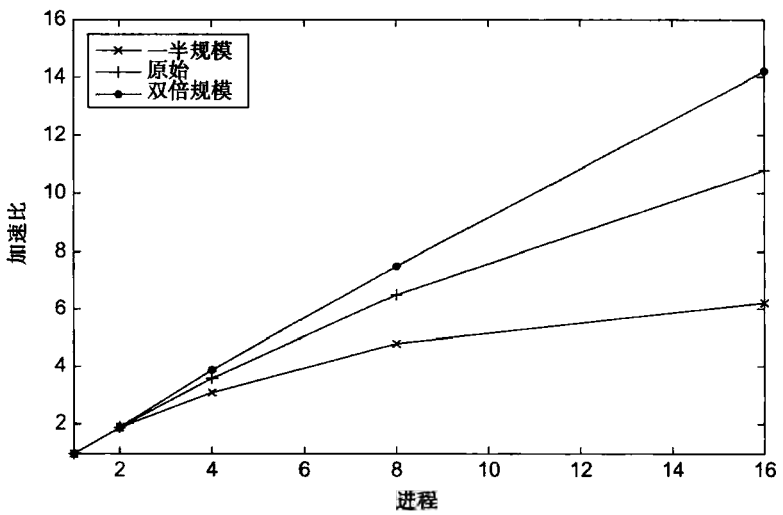
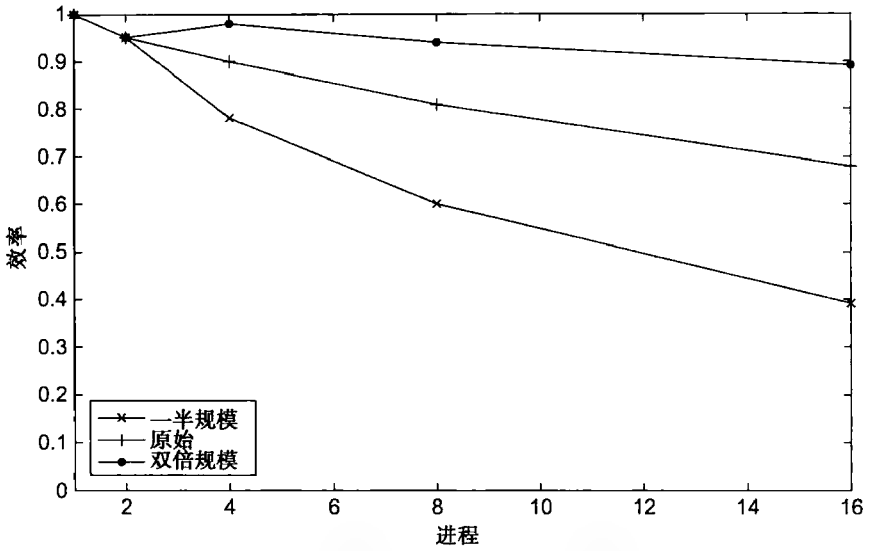



图 2-18　不同问题规模下并行程序的加速比




[60] 图 2-19 不同问题规模下并行程序的效率

### 2.6.2 阿姆达尔定律

回到20 世纪 60 年代，吉恩·阿姆达尔观察［2］到一个著名的定律：**阿姆达尔定律**（Amdahl's law）。该定律描述了这样的一个事实：大致上，除非一个串行程序的执行几乎全部都并行化，否则，不论有多少可以利用的核，通过并行化所产生的加速比都会是受限的。假设，一个串行程序中，可以并行化其中的90%。进一步假设，并行化是“理想”的，也就是说，如果使用 $p$ 个核，则程序可并行化部分的加速比就是 $p$。若该程序串行版本运行时间 $T_{串行}=20$ 秒，则并行化后，其中的可并行部分（即 90% 的部分）的运行时间就是 $(90\% \times T_{串行})/p = 18/p$ 秒，不可并行化部分的运行时间为 $10\% \times T_{串行}=2$ 秒。那么，程序并行版本的全部运行时间为：

$$T_{串行} = (90\% \times T_{串行})/p + 10\% \times T_{串行} = 18/p + 2$$

加速比为：

$$S = \frac{T_{串行}}{(90\% \times T_{串行})/p + 10\% \times T_{串行}} = \frac{20}{18/p + 2}$$

随着 $p$ 的增加，程序并行部分的运行时间 $(90\% \times T_{串行})/p = 18/p$ 会越来越趋向于0。但是，程序并行版本的总运行时间不可能小于 $10\% \times T_{串行}=2$ 秒。因此，加速比

$$S \leqslant \frac{T_{串行}}{10\% \times T_{串行}} = \frac{20}{2} = 10$$

也就是说，$S \leqslant 10$。这告诉我们这样一个事实：即使拥有 1000 个处理器核、即使对程序中 90% 的可并行部分完美地实现了并行化，也不可能得到比 10 更好的加速比。

考虑更一般的情况，串行程序中如果有比例为 $r$ 的部分不可并行化，则根据阿姆达尔定律，能达到的最好加速比趋近于 $1/r$。在上面的例子中，$r = 1 - 90\% = 10\%$，所以达不到比 10 更好的加速比。因此，如果 $r$ 代表串行程序的“天然串行”部分，即无法并行化部分所占的比例，我们不可能获得好于 $1/r$ 的加速比。即使 $r$ 非常小，例如，只有 1/100，即使我们使用一个拥有上千个核的系统，也不可能获得好于 100 的加速比。。

这听起来非常令人气馁，感觉即使做了这么多工作，却也是徒劳。但是，实际情况却并非如此，有这样几个理由可以让我们不用过多地考虑阿姆达尔定律所带来的影响。首先，阿姆达尔定
[61] 律中并没有考虑问题的规模。对于许多问题而言，当它的规模增加时，程序不可并行部分的比例

却在减小（更为形式化及具体的数学描述，参见 Gustafson 定律［25］）。其次，科学家和工程师所用的程序中，有成百上千个在分布式内存系统中获得了极大的加速比。最后，小的加速比难道就意味着很糟糕吗？在许多情况下，得到 5 ~ 10 的加速比就已经足够了，特别是当你不用费很大力气去开发并行程序的时候。

### 2.6.3　可扩展性

“可扩展”这个词有各种各样不正式的描述。事实上，我们已经在前文中运用了很多次这个名词。粗略地讲，如果一个技术可以处理规模不断增加的问题，那么它就是可扩展的。但是，对于并行程序的性能而言，可扩展性有一个更为正式的定义。假设我们运行一个拥有固定进程或线程数目的并行程序，并且它的输入规模也是固定的，那么我们可以得到一个效率值 $E$。现在，我们增加该程序所用的进程/线程数，如果在输入规模也以相应增长率增加的情况下，该程序的效率值一直都是 $E$，那么我们就称该程序是**可扩展的**。

举个例子，假设程序串行版本的运行时间是 $T_{串行} = n$，其中，$T_{串行}$的单位是微秒，$n$ 可看做是问题的规模。并且，假设程序并行版本的运行时间是 $T_{串行} = n/p + 1$。那么，效率值就为：

$$E = \frac{n}{p\ (n/p + 1)} = \frac{n}{n + p}$$

如果程序是可扩展的，以 $k$ 为倍率提高进程/线程的数目，希望在效率值 $E$ 不变的情况下，找到问题规模的增加比例 $x$。增加进程/线程的个数，使之变为 $kp$，问题的规模相应增加到 $xn$，通过解方程来求取 $x$ 的值：

$$E = \frac{n}{n + p} = \frac{xn}{xn + kp}$$

如果 $x = k$，那么 $xn + kp = kn + kp = k(n + p)$，则：

$$\frac{xn}{xn + kp} = \frac{kn}{k(n + p)} = \frac{n}{n + p}$$

换句话说，只有当问题规模增加的倍率，与进程/线程数增加的倍率相同时，效率才会是恒定的，从而程序是可扩展的。

对于可扩展性的描述，某些情况下会有一些特别的称谓。如果在增加进程/线程的个数时，可以维持固定的效率，却不增加问题的规模，那么程序称为**强可扩展的**（strongly scalable）。如果在增加进程/线程个数的同时，只有以相同倍率增加问题的规模才能使效率值保持不变，那么程序就称为**弱可扩展的**（weakly scalable）。我们前面举的例子就是弱可扩展的。 62

### 2.6.4　计时

在 2.6.3 节中，也许你会问这样的一个问题：$T_{串行}$与 $T_{并行}$到底是怎么得到的？其实，有很多种不同的方法获得这两个值。对于并行程序，详细的细节可能要取决于 API。然而，通过一些普遍的观察，或许能让这件事情变得稍微简单一些。

第一，至少有两个不同的理由让我们考虑计时（taking timings）。在程序的开发过程中，我们也许会通过计时来确定程序的运行情况是否是我们想要的那样。比如说，对于一个分布式内存程序，我们感兴趣的可能是，处理器为等待消息花了多少时间。如果说时间很长，那么大致可以肯定的是，我们的实现或者设计中存在某些问题。另一方面，一旦我们完成了程序的开发，我们通常想知道这个程序的性能到底怎么样。可能令你惊讶的是，其实对于这两种计时，我们采用的方法常常是不同的。对于第一种计时，通常需要非常详细的信息，可能需要知道程序在每一部分分别运行了多少时间，从而知道瓶颈在哪里。而对于第二种计时，我们只需要一个简单的数值。我们将要讨论的是第二种类型的计时。对于第一种类型，可以通过查看习题 2.22 来对其中的一些

问题做大致了解。

第二，我们通常对于程序从开始到结束之间的时间并不感兴趣，我们一般只对程序的某个部分感兴趣。比如说，编写一个冒泡排序的程序，我们可能只关心排序到底花了多少时间。而对于数据的读取和输出所花的时间，我们没有必要知道。因此可能无法使用类似于 UNIX shell 命令中的 time 命令，因为该命令告诉我们的是一个程序从开始到结束的运行时间。

第三，我们通常也对所谓的“CPU”时间不感兴趣。这是由标准 C 函数 clock 所提供的时间，代表的是程序执行代码的总时间。这个时间包括执行我们所写代码的时间、执行 pow 或 sin 等库函数的时间、调用系统函数（如 printf 和 scanf）所花的时间等。它不包括程序空闲状态的时间，这是它的一个问题。例如，在一个分布式内存程序中，一个进程若调用接收函数，它可能需要等待与它配对的另一个进程执行相应的发送操作，此时操作系统会使接收进程在等待的过程中处于睡眠状态。这个空闲时间并不会计入 CPU 时间中，因为此时没有任何函数被活动的进程调用。然而，在对整体运行时间的估计中，空闲时间是应该记录进来的，因为它确实是程序的真实开销的一部分。如果每次执行程序时，进程都必须等待一段时间，那么忽略等待时间将会得到错误的程序实际运行时间。

因此，当你在阅读一篇讲述并行程序运行时间的文章时，文章中的运行时间通常指的是“墙上时钟”时间。这个时间指的是代码从开始执行到执行结束的总耗费时间，这可能是我们比较关
63 心的。如果用户可以观察到程序的执行，那么在程序开始执行的那一刻，他按下了秒表的开始键；在程序停止执行的那一刻，他按下了结束键，这就是整个的“墙上时钟”时间。当然了，他看不到代码的执行，但是他可以修改源代码，像如下所示的这样：

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

Get_current_time()是一个虚构的函数，返回的是从过去固定的某一个时刻起，时间流逝的秒数。这里，它只是个抽象的表示。在实际的程序中，该函数的使用取决于具体的 API。例如，MPI 中的 MPI_Wtime 函数，共享内存编程 OpenMP API 中的函数是 omp_get_wtime。这两个函数返回的都是墙上时钟时间，而不是 CPU 时间。

关于计时函数，可能会有所谓的**分辨率**（resolution）问题。分辨率是指计时器的时间测量单位，是计时器在计时的过程中最短的非零时间跨度。有些计时器的分辨率为毫秒，当一条指令的运行时间少于 1 纳秒时，可能导致当计时器开始显示非零时间时，程序已经执行了上百万条指令。许多 API 提供报告计时分辨率的函数，另一些 API 则要求一个计时器必须有一个特定的分辨率。不管是哪种情况，作为编程人员，都需要检查这些值。

当我们对并行程序计时时，我们需要对计时方式有适当的了解。在下面的例子中，我们希望计时的代码段可能会被多个进程/线程执行，这将导致最后输出 $p$ 个运行时间。

```
private double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

然而，我们通常感兴趣的都是单个时间，即从第一个进程/线程开始执行，到最后一个进程/线程

结束之间花费的时间。一般很难精确地观察这个过程，因为可能一个进程的时钟和另一个进程的时钟之间并不保持一致。所以，我们通常寻找一个折中方案，代码如下所示。

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n", global_elapsed);
```

64

在这个例子中，我们首先执行一个**路障**（barrier）函数，同步所有的进程/线程。我们希望所有的进程/线程同时从路障函数调用的地方返回，但是通常只能保证当第一个进程/线程从该函数返回时，所有的进程/线程已经开始调用该函数。然后，像前面的例子一样执行代码，并且每个进程/线程记下它所耗费的时间。接着，所有的进程/线程调用一个全局最大值函数，该函数返回各个进程/线程所耗费时间的最大值，由 0 号进程/线程将该值打印出来。

我们还需要考虑计时的易变性（variability）。当多次执行同一个程序时，每次运行所花费的时间可能是不同的。即使我们每次都用同样的输入条件并在同样的系统上运行程序，这个情况仍然存在的。看起来最好的处理办法是采用运行的平均时间或者运行时间的中位数。但是，不可能由于某些外部事件，使程序的运行时间少于它可能的最短运行时间。所以，我们通常报告的是最短运行时间，而不是平均运行时间或者中位数时间。

如果在每个核中运行多于一个的线程，这会显著增加计时的易变性。更重要的是，如果在每个核中运行多个线程，系统将不得不花费多余的时间去进行调度，调度开销也会计入总的运行时间。因此，很少在一个核上运行多个线程。

最后，既然我们的程序不是为高性能 I/O 所设计的，我们通常不在报告的运行时间中包括 I/O 时间。

## 2.7　并行程序设计

如果已经有了一个串行程序，如何使之并行化呢？在一般情况下，需要将工作进行拆分，让其分布在各个进程/线程中，使每个进程所获得的工作量大致相同，并且使通信量最小。所以很多情况下，我们还需要安排进程/线程之间的同步与通信。不幸的是，我们无法通过执行固定的步骤达到上述目标。否则，我们就可以编写一个程序，用它去转换任意一个串行程序，使之并行化。但就像第 1 章提到的那样，尽管我们做了大量的工作并且有了一些进展，但这仍然是一个没有通用解决办法的问题。

65

然而，Ian Foster 在他的在线电子书《Designing and Building Parallel Programs》[19] 中，给出了一个并行化步骤：

1）划分（partitioning）。将要执行的指令和数据按照计算部分拆分成多个小任务。这一步的关键在于识别出可以并行执行的任务。

2）通信（communication）。确定上一步所识别出来的任务之间需要执行那些通信。

3）凝聚或聚合（agglomeration or aggregation）。将第一步所确定的任务与通信结合成更大的任务。例如，如果任务 A 必须在任务 B 之前执行，那么将它们聚合成一个简单的复合任务可能更为明智。

4）分配（mapping）。将上一步聚合好的任务分配到进程/线程中。这一步还要使通信量最小化，使各个进程/线程所得到的工作量大致均衡。

这个步骤有时候称为 Foster 方法。

### 实例

来看一个简单的实例。假设有一个程序，该程序生成大量的浮点数并将其存储在一个数组里。为了对数据分布有一个更为直观的感受，可以画一个数据的直方图。回想制定直方图的过程，首先简单地将数据的范围划分成几个同等大小的子区间，或称为桶。然后确定每个桶中数据的个数，并绘制一个直方图，以展示各个桶中数据的数目。作为一个简单的例子，假设数据为：

1.3，2.9，0.4，0.3，1.3，4.4，1.7，0.4，3.2，0.3，4.9，2.4，3.1，4.4，3.9，0.4，4.2，4.5，4.9，0.9。

这些数据都分布在 0 ~ 5 之间。如果我们使用五个桶，那么直方图就会像图 2-20 所描绘的那样。

#### 串行程序

编写一个生成直方图的串行程序非常容易。我们只需要确定有几个桶，以及每个桶中数据的个数，然后打印出直方图即可。由于不考虑 I/O 的影响，所以我们只将注意力放在前面两个步骤上。所以，输入部分为：

1）数据的个数 data_count。

2）一个大小为 data_count 的浮点数数组 data。

3）包含最小值的桶中的最小值 min_meas。

4）包含最大值的桶中的最大值 max_meas。

66 5）桶的个数 bin_count。

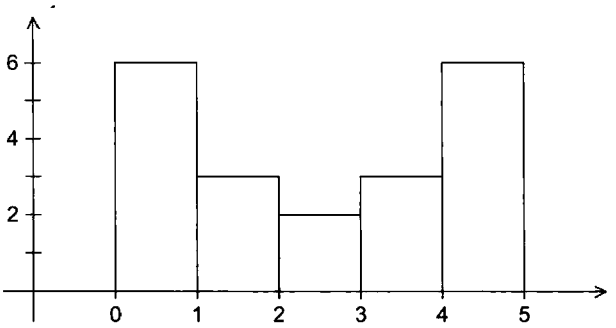



图 2-20　一个直方图

输出部分为一个数组，数组中的每个元素对应于桶中数据的个数。为了更精确地描述整个过程，使用以下的数据结构：

```
- bin_maxes        一个大小为 bin_count 的浮点数数组
- bin_counts       一个大小为 bin_count 的整数数组
```

bin_max 数组存储的是每个桶的上界，而 bin_counts 数组存储的是落在每个桶里的数据的个数。为了更加清楚，我们定义：

```
bin_width = (max_meas - min_meas)/bin_count
```

则 bin_maxes 数组的初始化过程为：

```
for (b = 0; b < bin_count; b++)
   bin_maxes[b] = min_meas + bin_width*(b+1);
```

按照习惯，落在桶 $b$ 中所有数据的取值（measurement）都应该在下面这个范围里：

```
bin_maxes[b-1] <= measurement < bin_maxes[b]
```

当然，这个公式不包括 $b=0$ 的情况。桶 0 的范围为：

```
min_meas <= measurement < bin_maxes[0]
```

这就意味着，我们需要将桶 0 视为特殊情况处理，这个工作其实并不复杂。

一旦初始化了 bin_maxes 数组，并将 bin_counts 的每个元素值都设为 0，就可以用下面的伪代码来得到计数值：

```
for (i = 0; i < data_count; i++){
   bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);
   bin_counts[bin]++;
}
```

函数 Find_bin 返回的是 data[i]属于的那个桶号。这是个简单的线性查找函数：线性查找 bin_maxes 数组，直到发现满足下面条件的桶 $b$：

```
bin_maxes[b-1] <= data[i] < bin_maxes[b]
```

（这里，bin_maxes[-1]的值为 min_meas。）如果桶的数目不是很多，那么使用线性查找就可 67
以了。但是一旦桶的数目过多，运用二分查找将会使性能好很多。

**并行化串行程序**

如果 data_count 比 bin_count 大很多，即使在 Find_bin 函数中采用二分查找，整个代码的大部分工作量就会集中在确定 bin_counts 数组中各个元素大小的循环中。所以，并行化的重点应该放在这个循环上，我们将运用 Foster 方法来实现并行化。首先要注意的是，Foster 方法中每一步骤的输出，并不是唯一确定的。所以在任何一步中，如果你想到了不同的方法，也没什么好惊讶的。

第一步，识别出两类任务：找出 data 数组中元素所属于的那个桶；该桶对应的 bin_counts 数组元素加 1。

第二步，计算桶号，将桶号对应的 bin_counts 数组元素加 1，它们之间一定存在通信。如果用椭圆来表示任务，用箭头来表示任务之间的通信，那么会得到一个类似图 2-21 这样的通信图。这里，标记为 data[i]的任务决定了 data[i]所属的桶，标记为 bin_counts[b]++的任务将 bin_counts[b]的值加 1。

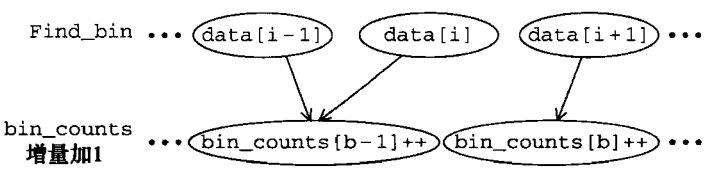



图 2-21　Foster 方法最开始的两个阶段

对于 data 数组中的每个固定元素，任务“找出元素所落在的桶的桶号”与任务“将 bin_counts[b]值加 1”可以整合在一起，因为后者只有在前者完成之后才能进行。

然而，当进行到最后的分配步骤时，我们发现，如果两个进程或线程分配到的 data 数组元素属于同一个桶 b 时，它们都将执行 bin_counts[b]++这条语句。如果 bin_counts[b]是

共享的（例如，bin_counts 数组存储在共享内存里），这将导致发生竞争。如果 bin_counts 数组被拆分到各个进程/线程中，更新该数组中的元素就需要进程/线程之间的通信。一种解决办法是，存储 bin_counts 的多个“本地”副本，所有 Find_bin 进行后，对本地副本中的值加 1。

如果桶的个数，即 bin_count 的值不是非常大，那么不会产生什么问题。所以我们暂且就使用这种方法，因为无论对于共享内存系统，还是分布式内存系统，这种方法都适用。

在这种设置下，需要更新任务图，使第二个任务集合能够增加 loc_bin_cts[b]的值。我们还需要增加第三个任务集合，将各个 loc_bin_cts[b]加起来，从而得到 bin_counts[b]的值，如图 2-22 所示。从图 2-22 中可以看到，如果对每个进程/线程都建立了一个 loc_bin_cts
68 数组，那么任务可以分为两组：

1）将 data 数组的元素分配到各个进程/线程，以使每个进程/线程得到的元素个数大致相同。

2）每个进程/线程根据分配到的元素，更新 loc_bin_cts 数组中的值。

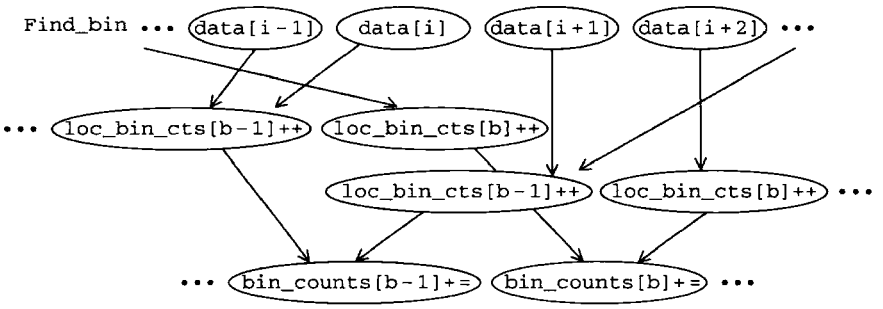



图 2-22　任务与通信的另一种定义方式

最后，需要将各个 loc_bin_cts[b]都加起来，从而得到 bin_counts[b]的值。如果进程/线程数和桶数都很小，那么所有的加法操作都可以交给单个进程/线程去完成。如果桶数远远大于进程/线程数，那么可以将桶拆分到各个进程/线程中，就像我们拆分 data 数组那样。如果进程/线程数较大，可以采用第 1 章介绍的树形结构的全局求和法的类似方法。唯一与之前介绍的全局求和法不同的是，发送进程/线程发送的是一个数组，接收进程/线程所接收和加起来的也是个数组。图 2-23 显
69 示了一个具有 8 个进程/线程的例子。最

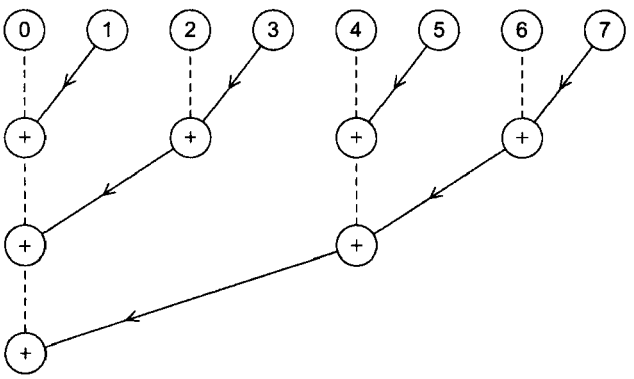



图 2-23　本地数组相加

上面一层中的每个圆圈代表的是一个进程/线程。在第一层与第二层之间，偶数编号的进程/线程可以得到奇数编号的进程/线程所提供的 loc_bin_cts 数组，并将得到的值加到自己的值上。在第二层与第三层之间，编号能够被 4 整除的进程与编号不能被 4 整除的进程重复上面类似的操作。整个过程不断重复，直到 0 号进程/线程计算出 bin_counts 数组的值。

## 2.8　编写和运行并行程序

过去，几乎所有的并行程序的开发都是用 vi 或 Emacs 之类的文本编辑器完成的。程序通过命令行或者编辑器来编译和运行程序。同样，调试器也一般通过命令行来操作。现在，我们可以

直接使用集成开发环境（Integrated Development Environment，IDE）来开发程序，比如微软的开发环境、Eclipse，以及其他开发环境。参见［16，38］。

在规模较小的共享内存系统中，运行单个的操作系统副本，在可用核之间进行线程调度。在这些系统上，共享内存程序通常由 IDE 或者命令行启动。一旦启动后，程序一般通过控制台（console）和键盘来进行标准输入（stdin），并将输出的数据输出到标准输出（stdout）和标准错误输出（stderr）中。在更大的系统中，可能会有一个批处理调度器，通过调度器，用户可以请求一定数量的核，并指定可执行路径和输入/输出的位置（通常是二级存储器上的文件）。

在典型的分布式内存系统和混合系统中，由一台宿主计算机负责用户间的节点分配工作。有些系统是纯粹的批处理系统，有点像共享内存批处理系统。而另一些系统会允许用户退出节点，交互运行任务。由于任务的启动通常会涉及与远程系统的通信，实际的启动一般在一个脚本里完成。例如，MPI 程序通常由叫做 mpirun 或 mpiexec 的脚本来启动。

RTFD 通常翻译成“阅读好的文档”（read the fine documentation）。

## 2.9　假设

如前所述，我们将重点关注同构 MIMD 系统，即系统中所有节点的结构相同，程序结构是 SPMD 的。因此，我们编写的是一个拥有不同分支的单个程序，并假定各个核的结构相同，但异步操作。我们还假定，在单个核上最多只运行一个进程或线程，并经常使用静态进程或线程。换句话说，我们会在差不多同一时间开启所有的进程或线程，并且在它们完成执行时，在差不多同一时间终止它们。 [70]

某些用于并行系统的应用程序编程接口定义了新的编程语言。但大多数 API 是对现成编程语言的扩展，可以通过库函数（如消息传递函数）实现扩展；也可以扩展用于串行版本的编译器。本节关注后一种方法。我们将使用的是 C 语言的并行扩展。

当我们想显式地编译和运行程序时，使用 UNIX shell 的命令行形式，或者 gcc 编译器，或者 gcc 的某种扩展（如 mpicc）。而且，也用命令行来启动程序。例如，如果我们想显示 Kernighan 和 Ritchie 编写的“hello，world”程序［29］的编译与执行，会出现的信息如下：

```
$ gcc -g -Wall -o hello hello.c
$ ./hello
hello, world
```

其中，$ 是 shell 的标志符。我们通常使用如下的编译器选项：

- -g　允许使用调试器。
- -Wall　显示警告。
- -o <outfile>　编译出的可执行文件的文件名为 outfile。
- 在对程序计时时，我们用 -O2 选项来告诉编译器对代码进行优化。

在大部分系统中，用户的目录或文件夹在默认情况下是不出现在用户的执行路径上的。所以，我们会在可执行文件的文件名前加上 ./来给出可执行文件的路径。

## 2.10　小结

本章讨论了许多内容，一个完整的总结会占用许多页。下面我们尽量精简地做个总结。

### 2.10.1　串行系统

本章从讨论传统的串行硬件和串行软件开始。计算机硬件的标准模型是**冯·诺依曼结构**，由执行计算的**中央处理单元**（CPU），以及存储数据及指令的**主存**组成。CPU 与主存的分离通常称

为**冯·诺依曼瓶颈**，因为这会限制指令执行的速率。

计算机上最重要的软件可能就是**操作系统**，它管理计算机的资源。大部分现代操作系统都是**多任务的**。即使硬件上没有多个处理器或者核，通过对执行程序进行快速切换，操作系统可以制
71 造出多个程序同时运行的假象。一个正在执行的程序称为一个**进程**。由于一个进程或多或少是独立运行的，它会包含许多相关的数据和信息。而**线程**由进程启动，它不需要独立包含相关数据及信息，所以线程的终止与开启比进程要快得多。

在计算机科学中，**缓存**（cache）是存储单元的一种。与其他存储单元相比，缓存的访问更快。CPU 缓存是在 CPU 寄存器与主存之间的中间存储器，主要目的是降低主存访问的延迟。缓存利用了一些局部性原理，即与近期访问的数据相邻的数据可能会在不久的将来被访问。当一条指令或数据被访问时，如果它已经在缓存里了，我们就称之为缓存**命中**。如果不在缓存里，则称为**缺失**。缓存由计算机硬件直接管理，所以程序员只能间接控制缓存。

主存也可以被看做是二级存储器的缓存。由硬件和操作系统通过**虚拟内存**来管理。一般情况下，程序中所有的指令和数据不会都存储在主存里，在主存中只保留活动的部分，而其他部分则存储在二级存储的**交换空间**里。与 CPU 缓存一样，虚拟内存对连续的数据或指令组成的块（我们称为**页**）进行操作。虚拟内存不采用内存的物理地址进行编址，而采用**虚拟地址**，与实际的物理地址相独立。物理地址与虚拟地址的对应关系存储在主存的页表里。虚拟地址和页表的结合让系统可以在内存的任何地方存储程序的数据与指令。因此，如果两个不同的程序使用了相同的虚拟地址也没有关系。由于使用了页表，每次程序要访问主存时，需要两次访问主存：一次是获取页表项，从而知道要虚拟地址所对应的物理内存的位置；另一次是对需要访问的数据进行实际的访问。为了避免这个问题（主存访问的次数增加），CPU 有一个特殊的页表缓存，称之为转译后备缓冲器（TLB），存储最近使用到的页表项。

指令级并行（ILP）使单进程可以同时执行多个指令。有两种主要的 ILP：**流水线**和**多发射**。对于流水线，处理器的功能单元被依次排列，其中一个的输出作为另一个的输入。当一个数据在第二个功能单元内处理时，另一个数据就能在第一个处理单元内处理。对于多发射，同类型的功能单元会被复制，处理器可以同时在单个程序中执行多条不同的指令。

与同时执行指令不同，**硬件多线程**同时运行不同的线程。有多种不同的方法可以实现硬件多线程。然而，所有这些方法都是通过快速在线程之间进行切换来试图使处理器尽可能地忙碌。当
72 一个线程发生**阻塞**并在执行指令之前必须等待时（如为了等待访存完成），硬件多线程显得尤其重要。在**同步多线程**机制里，不同的线程可以在一个多发射处理器内同时使用多个功能单元。由于线程是由多个指令组成的，有时我们会说**线程级并行**（TLP）比 ILP 更**粗粒度**。

### 2.10.2 并行硬件

指令级并行和线程级并行在底层提供并行性，典型情况下，它们由处理器和操作系统来控制，而不是由程序员直接控制。而本书的目标是：如果并行性对于程序员是可见的，那么硬件就是并行的，程序员可以通过修改代码来开发并行性。

通常用 Flynn **分类法**对并行硬件进行分类，通过系统可以处理的指令流数目和数据流数目来区别各个分类。冯·诺依曼系统拥有单个的指令流和单个的数据流，所以它是**单指令单数据流**（SISD）系统。

**单指令多数据流**（SIMD）系统在任一时间执行一条指令，但是该指令可以对多个数据项进行操作。这些系统经常按锁步执行指令：第一条指令同时应用在所有的数据项上，然后是第二条指令，以此类推。这种并行系统通常使用**数据并行**程序，在数据并行程序里，将数据划分给各个处理器，各个数据由相同的指令序列来处理。**向量处理器**与**图形处理器单元**（GPU）一般划分在

SIMD 系统里，尽管最近的 GPU 也有多指令多数据流系统的特点。

在 SIMD 系统里，对分支指令的处理主要是：让那些没有对数据项使用分支指令的处理器处于空闲状态。这种行为使得 SIMD 系统不适合**任务并行**。在任务并行里，每个处理器执行一个不同的任务。SIMD 系统甚至不适合拥有多个条件分支的数据并行的操作。

对于**多指令多数据流**（MIMD）系统，正如其名，系统同时执行多个指令流，每个指令流有自己的数据流。实际上，MIMD 系统是多个自主处理器的集合，每个处理器可以按照自己的方式运行。不同 MIMD 系统的主要区别在于：它们是**共享内存**系统，还是**分布式内存**系统。在共享内存系统里，每个处理器或者核可以直接对所有的内存单元进行访问；而在分布式内存系统里，每个处理器有它自己的私有内存。大多数大型 MIMD 系统其实是**混合**系统，是由多个相对小的共享内存系统通过网络连接来实现的。在这些系统里，单个的共享内存系统有时称为节点。有些 MIMD 系统是**异构**系统，处理器拥有不同的性能。例如，拥有一个传统 CPU 和一个 GPU 的系统就是一个异构系统。而如果一个系统中的所有处理器都是相同结构的，那么就是**同构**的。 [73]

在共享内存系统中，处理器与内存之间会有不同的连接方式，同样，在分布式内存或者混合系统中，处理器之间也有多种不同的连接方式。共享内存系统中，最普遍使用的连接方式是**总线**和**交叉开关矩阵**。而分布式内存系统有时会采取**直接**连接的方式，如二维环面网格和超立方体，有时也采用交叉开关矩阵和多级网络这类**间接**连接方式。网络的性能通常由**等分宽度**或**等分带宽**来评估，用来衡量网络支持多少同时通信。对于节点间的单个通信，我们一般讨论通信的**延迟**和**带宽**。

共享内存系统的一个潜在问题是**缓存一致性**。相同变量可以存储在两个不同核的缓存中。如果一个核更新了变量值，另一个核却不知道该变量已经被改变了。有两种主要方法可以保证缓存的一致性：**监听**和使用**目录**。监听需要依靠互连结构从一个缓存控制器向其他缓存控制器广播信息的能力。目录是特殊的分布式数据结构，它存储每一条缓存行的信息。缓存一致性引出了共享内存编程的另一个问题：**伪共享**。在一个核更新了一个缓存行上某变量的值后，如果另一个核想要访问同一缓存行上的另一个变量，那么它不得不访问主存，因为缓存一致性的单位是行。换句话说，第二个核只知道它要访问的缓存行被更新了，它并不知道它要访问的变量其实并没有改变。

### 2.10.3　并行软件

本节，我们关注同构 MIMD 系统的软件开发。此类系统的大部分程序是单个程序，并通过分支语句实现并行。这种程序通常称为单程序多数据流（SPMD）程序。在共享内存程序中，将正在执行的任务实例称为线程；在分布式内存程序里，我们称它们为进程。

除非问题是**易并行的**，并行程序的开发至少需要考虑进程或线程间的**负载平衡**、**通信**，以及**同步**。

在共享内存程序中，单个线程可以拥有**私有**和**共享**内存，通信通常通过**共享变量**来完成。当处理器异步执行时，会存在**不确定性**问题。就是说，对于一个给定的输入，程序的两次运行会有不同的结果。这会成为一个严重的问题，特别是在共享内存程序里。如果不确定性是由两个线程试图访问相同资源而引起的，并且它会导致一个错误，那么该程序就称为含有一个**竞争条件**。竞争条件最可能存在的地方是**临界区**。临界区是一个代码块，在任意时间只能有一个线程可以执行该代码块。在大部分的共享内存 API 里，临界区的**互斥**可由称为**互斥锁**或**互斥量**（mutex）的对 [74]
象来实现。临界区应该越小越好，因为互斥量保证了在任意时间只允许一个线程在临界区内执行，这会使代码串行化。

对于共享内存程序的第二个潜在问题是**线程安全性**。当一个代码块被多个线程运行并且运行

正确时，称之为**线程安全的**。为串行程序所写的函数会毫不知情地使用共享数据，例如，静态变量。在多线程程序里，这样使用共享数据会导致错误。这种函数不是线程安全的。

分布式内存系统编程使用最多的 API 是**消息传递**。在消息传递 API 中，至少有两个不同的函数：一个**发送**函数和一个**接收**函数。当进程需要通信时，一个进程调用发送函数，另一个调用接收函数。这些函数可以有各种可能的行为。例如，发送函数**阻塞**或等待，直到对应的接收函数开始运行，或者，消息传递软件将消息数据复制到自己的存储空间内，发送进程可以在接收开始之前就返回。最一般的接收行为是，直到消息接收前都保持阻塞状态。使用最多的消息传递系统叫做**消息传递接口**（MPI）。它在简单的发送和接收的基础上提供了许多功能。

分布式内存系统还可以用**单向通信**来进行编程。它使得进程可以访问属于其他进程的数据。而**划分全局地址空间**语言在分布式内存系统中提供了一些共享内存功能。

### 2.10.4 输入和输出

在一般的并行系统里，多个核可以访问多个二级存储设备。但我们不会编写使用该功能的程序。相反，我们会编写的程序是，其中的一个进程或线程可以访问标准输入（stdin），所有进程可以访问标准输出（stdout）和标准错误输出（stderr）。然而，由于不确定性，除了调试输出外，通常只让一个进程或线程访问标准输出（stdout）。

### 2.10.5 性能

如果用 $p$ 个进程或线程来运行一个并行程序，并且每个核中至多只有一个进程/线程，那么理想状态下，并行程序运行速度应该是串行程序的 $p$ 倍。这称为**线性加速比**，但实际上，线性加速比很难实现。如果我们用 $T_{串行}$ 来表示串行程序的运行时间，用 $T_{并行}$ 来表示并行程序的运行时间，那么并行程序的**加速比** $S$ 和**效率** $E$ 可由以下公式分别得到：

75

$$S = \frac{T_{串行}}{T_{并行}}$$

$$E = \frac{T_{串行}}{p \times T_{并行}}$$

所以，对于线性加速比，$S = p$，$E = 1$。实际上，我们得到的结果几乎都是 $S < p$，$E < 1$。如果我们固定问题的规模，那么当 $p$ 增加时，$E$ 通常会减小。如果我们固定进程/线程的数量，当我们增大问题的规模时，$S$ 和 $E$ 都会增加。

**阿姆达尔定律**提供了一个并行程序可以得到的加速比的上界：如果原有的串行程序中无法并行化部分在整个程序中所占的比例是 $r$，那么无论使用多少进程/线程，得到的加速比都无法超过 $1/r$。但实际上，很多并行程序都能得到很好的加速比。产生这一矛盾的原因可能是：当问题规模增大时，无法并行的部分相对于可并行部分所占的比例会降低。

**可扩展性**这一术语有很多种解释。一般而言，如果一个技术可以处理任意增长的问题规模，那么我们就称其为可扩展的。正式的表述是：一个并行程序，如果问题的规模与进程/线程数都以一定的倍率增加，而效率保持一个常数值，那么该并行程序就是可扩展的。如果问题规模保持不变，那么该程序就称为**强可扩展的**。如果问题规模的大小随着进程/线程数的增长等倍率增长，那么该程序就称为**弱可扩展的**。

为了确定 $T_{串行}$ 和 $T_{并行}$，我们在源代码里通常要调用计时函数。我们希望计时函数提供的是**墙上时钟**时间，而不是 CPU 时间。这主要是因为，当 CPU 空闲时，程序仍然可能是“活动”的（如等待一条消息）。为了得到并行计时时间，在计时器开始之前，通常需要同步进程/线程，并在计时后，找出进程/线程中的最长运行时间。因为系统的易变性，一般要多次运行程序，并且记录下最短的运行时间。为了降低易变性，提高总体的运行时间，通常在每个核上最多运行一个线程。

### 2.10.6　并行程序设计

**Foster 方法**给出了设计并行程序的一系列步骤。这些步骤为：**划分**问题并识别任务；在任务中识别要执行的**通信**；**凝聚**或**聚合**任务使之变成较大的组任务；将聚合任务**分配**给进程/线程。

### 2.10.7　假设

我们所关注的主要是在共享内存与分布式内存 MIMD 系统上的并行程序开发。我们编写使用静态进程/线程的 SPMD 程序。这些进程/线程在程序开始执行时生成，直到程序结束后才关闭。我们还假设系统中的每个核上最多运行一个进程或线程。 [76]

## 2.11　习题

2.1　当讨论浮点数加法时，我们简单地假设每个功能单元都花费相同的时间。如果每个取命令与存命令都耗费 2 纳秒，其余的每个操作耗费 1 纳秒。

a. 在上述假设下，每个浮点数加法要耗费多少时间？

b. 非流水线 1000 对浮点数的加法要耗费多少时间？

c. 流水线 1000 对浮点数加法要耗费多少时间？

d. 如果操作数/结果存储在不同级的内存层级上，那么取命令与存命令所要耗费的时间可能会差别非常大。假设从一级缓存上取数据/指令要耗费 2 纳秒，从二级缓存上取数据/指令要耗费 5 纳秒，从主存取数据/指令要耗费 50 纳秒。当执行某条指令，取其中一个操作数时，发生了一次一级缓存失效，那么流水线会发生什么情况？如果又发生二级缓存失效，又会怎样？

2.2　请解释在 CPU 硬件里实现的一个队列，怎么使用可以提高写直达高速缓存（write - through cache）的性能。

2.3　回顾之前一个从缓存读取二维数组的示例。请问一个更大矩阵和一个更大的缓存是如何影响两对嵌套循环的性能的？如果 MAX = 8，缓存可以存储 4 个缓存行，情况又会是怎样的？在第一对嵌套循环中对 A 的读操作，会导致发生多少次失效？第二对嵌套循环中的失效次数又是多少？

2.4　在表 2-2 中，虚拟地址由 12 位字节偏移量和 20 位的虚拟页号组成。如果一个程序运行的系统上拥有这样的页大小和虚拟地址空间，这个程序有多少页？

2.5　在冯·诺依曼系统中加入缓存和虚拟内存改变了它作为 SISD 系统的类型吗？如果加入流水线呢？多发射或硬件多线程呢？

2.6　假设一个向量处理器的内存系统需要用 10 个周期从内存载入一个 64 位的字。为了使一个载入流的平均载入时间为一个周期载入一个字，需要多少个内存体（memory bank）？

2.7　请讨论 GPU 与向量处理器执行以下代码时的不同之处：

```
sum = 0.0;
for (i = 0; i < n; i++) {
   y[i] += a*x[i];
   sum += z[i]*z[i];
}
```

[77]

2.8　如果硬件多线程处理器拥有大缓存，并且运行多个线程，请解释为何该处理器的性能会下降。

2.9　在关于并行硬件的讨论中，用 Flynn 分类法来识别三种并行系统：SISD、SIMD 和 MIMD。我们讨论的系统中没有多指令流单数据流系统，或者称为 MISD 系统。那么，MISD 系统是如何工作的呢？请举例说明。

2.10　假设一个程序需要运行 $10^{12}$ 条指令来解决一个特定问题，一个单处理器系统可以在 $10^6$ 秒（大约 11.6 天）内完成。所以，一个单处理器系统平均每秒运行 $10^6$ 条指令。现在假设程序已经实现并行化，可以在分布式内存系统上运行。该并行程序使用 $p$ 个处理器，每个处理器执行 $10^{12}/p$ 条指令并必须发送 $10^9$（$p-1$）条消息。执行该程序时，不会有额外的开销，即每个处理器执行完所有的指令并发送完所有的消息之后，程序就完成了，而不会有由诸如等待消息等事件所产生的延迟。那么，

a. 假设发送一条消息需要耗费 $10^{-9}$ 秒。如果程序使用 1000 个处理器，每个处理器的速度和单个处理器运行串行程序的速度一样，那么该程序的运行需要多少时间？

b. 假设发送一条消息需要耗费 $10^{-3}$ 秒。如果程序使用 1000 个处理器，那么该程序的运行需要多少时间？

2.11 请写出不同的分布式内存互连形式的总链路数的表达式。

2.12 a. 除了没有循环链接（“wraparound” link），平面网格（planar mesh）和二维环面网格（toroidal mesh）是相似的。请问一个平面网格的等分宽度是多少？

b. 三维网格与平面网格是相似的，除了三维网格拥有深度这个特性外。请问一个三维网格的等分宽度是多少？

2.13 a. 请画出一个四维超立方体结构。

b. 请用超立方体的归纳定义来解释为何超立方体的等分宽度为 $p/2$。

2.14 为了定义间接网络的等分宽度，我们将处理器分为两组，每组拥有一半数量的处理器。然后，在网络的任意处移除链接，使两组之间不再连接。移除的最小链路数就是该网络的等分宽度。当我们对链路计数时，如果图中用的是单向链接，则两条单向链接算作一条链路。请说明一个 8 × 8 的交叉开
78 关矩阵的等分宽度小于或等于 8，并说明一个拥有 8 个处理器的 omega 网络的等分宽度小于或等于 4。

2.15 a. 假定一个共享内存系统使用监听缓存一致性协议和写回缓存。并且假设 0 号核的缓存里有变量 x，并执行赋值命令 x = 5。1 号核的缓存里没有变量 x。当 0 号核更新了 x 后，1 号核开始尝试执行 y = x。y 被赋的值是多少？为什么？

b. 假定上面的共享内存系统使用的是基于目录的协议，则 y 的值将是多少？为什么？

c. 你能否为前两部分中所发现的问题提出解决方案？

2.16 a. 假定一个串行程序的运行时间为 $T_{串行} = n^2$，运行时间的单位为毫秒。并行程序的运行时间为 $T_{并行} = n^2/p + \log_2(p)$。对于 $n$ 和 $p$ 的不同值，请写一个程序并找出这个程序的加速比和效率。在 $n$ = 10、20、40、…、320 和 $p$ = 1、2、4、…、128 等不同情况下运行该程序。当 $p$ 增加、$n$ 保持恒定时，加速比和效率的情况分别如何？当 $p$ 保持恒定而 $n$ 增加呢？

b. 假设 $T_{并行} = T_{串行}/p + T_{开销}$，我们固定 $p$ 的大小，并增加问题的规模。

- 请解释如果 $T_{开销}$ 比 $T_{串行}$ 增长得慢，随着问题规模的增加，并行效率也将增加。
- 请解释如果 $T_{开销}$ 比 $T_{串行}$ 增长得快，随着问题规模的增加，并行效率将降低。

2.17 如果一个并行程序所获得的加速比可以超过 $p$（进程或线程的个数），则我们有时称该并行程序拥有**超线性加速比**（superlinear speedup）。然而，许多作者并不将能够克服“资源限制”的程序视为是拥有超线性加速比。例如，当一个程序运行在一个单处理器系统上时，它必须使用二级存储，当它运行在一个大的分布式内存系统上时，它可以将所有数据都放置到主存上。请给出另外一个例子，说明程序是如何克服资源限制，并获得大于 $p$ 的加速比的。

2.18 请观察你在计算机科学导论课上编写的三个程序。这些程序中有哪些部分本来就是串行的？当问题规模增加时，串行部分工作所占的比例会减少吗？或者保持大致相同？

2.19 假定 $T_{串行} = n$，$T_{串行} = n/p + \log_2(p)$，时间单位为毫秒。如果以倍率 $k$ 增加 $p$，那么为了保持效率值的恒定，需要如何增加 $n$？请给出公式。如果我们将进程数从 8 加倍到 16，则 $n$ 的增加又是多少？该并行程序是可扩展的吗？

79 2.20 一个可以获得线性加速比的程序是强可扩展的吗？请解释。

2.21 Bob 有个程序，想对两组数据进行计时，input_data1 和 input_data2。为了在程序中加入计时函数前得到一些想法，他用两组数据和 UNIX 的 shell 命令 time，运行了程序：

```
$ time ./bobs_prog < input_data1

real 0m0.001s
user 0m0.001s
sys  0m0.000s
$ time ./bobs_prog < input_data2

real 1m1.234s
user 1m0.001s
sys  0m0.111s
```

Bob 用的时间函数的精度为毫秒。Bob 应该使用第一组数据和时间函数对他的程序进行计时吗？如果使用第二组数据呢？请分别解释使用和不使用的原因。

2.22　正如我们在习题 2.21 中所看到的，UNIX 的 shell 命令 time 报告用户时间、系统时间，以及“实际”时间或全部耗费的时间。假设 Bob 定义了以下这些可以被 C 程序调用的函数：

```
double utime(void);
double stime(void);
double rtime(void);
```

第一个函数返回的是从程序开始执行用户时间所耗费的秒数。第二个返回的是系统时间秒数，第三个是总时间秒数。大致上，用户时间主要耗费在不需要操作系统执行的用户代码和库函数上，如 sin 和 cos 函数。系统时间耗费在那些需要操作系统执行的函数上，如 printf 和 scanf 函数。

a. 这三个时间函数值的数学关系是什么样的？假定程序包含如下代码：

```
u = double utime(void);
s = double stime(void);
r = double rtime(void);
```

请写出 u、s 和 r 之间关系的表达式（可以假定忽略函数调用的时间花费）。

b. 在 Bob 的系统上，任何时候，如果一个 MPI 进程在等待消息，则它花费的时间不计入 utime 和 stime，而计入 rtime。请解释 Bob 是如何根据这些条件来确定一个 MPI 进程是否在等待消息上耗费了过多时间。 [80]

c. Bob 提供给了 Sally 他的计时函数。然而，Sally 发现在她的系统上，一个 MPI 进程在等待消息上的时间耗费是计入用户时间的。那么，Sally 可以用 Bob 的函数去判断一个 MPI 进程是否在等待消息上耗费了过多时间吗？请解释。

2.23　在我们应用 Foster 方法来构建直方图的过程中，我们实质上是用 data 数组的元素来识别聚合任务的。一个很明显的替代方法是，使用 bin_counts 数组的元素来识别聚合任务，所以一个聚合任务会由 bin_counts[b]的增加，和返回 b 的 Find_bin 函数的调用所组成。请解释为何这样的聚合可能存在问题。

2.24　如果你在第 1 章还没有完成，那么请试着编写树形结构的全局求和的伪代码，其作用是对 loc_bin_cts 数组的元素进行求和。请先考虑在共享内存的情况下该如何实现。接着考虑分布式内存的情况。在共享内存的情况下，哪些变量是共享的，哪些是私有的？ [81]

第3章
An Introduction to Parallel Programming

# 用 MPI 进行分布式内存编程

回想一下，大部分并行多指令多数据流计算机，都分为**分布式内存**系统和**共享内存**系统两种。从程序员的角度看，一个分布式内存系统由网络连接的核 - 内存对的集合组成，与核相关联的内存只能由该核访问。如图 3-1 所示。另一方面，从程序员的角度看，共享内存系统由核的集合组成，所有核都连接到一个全局访问的内存，且每个核可以访问内存的任意位置。如图 3-2 所示。本章将讨论如何使用**消息传递**来对分布式内存系统进行编程。

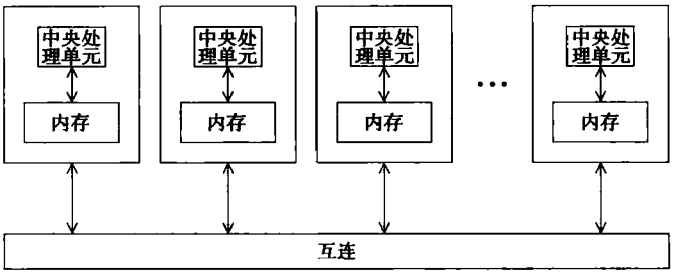



图 3-1　一个分布式内存系统

在消息传递程序中，运行在一个核 - 内存对上的程序通常称为一个**进程**。两个进程可以通过调用函数来进行通信：一个进程调用*发送*函数，另一个调用*接收*函数。我们将使用消息传递的实现称为消息传递接口（Message-Passing Interface，MPI）。MPI 并不是一种新的语言，它定义了一个可以被 C、C ++ 和 Fortran 程序调用的函数*库*。我们将学习 MPI 中的一些不同的发送与接收函数，还将学习一些可以涉及多于两个进程的“全局”通信函数。这些函数称为**集合**通信。在学习这些
83 MPI 函数的过程中，我们还会了解一些涉及编写消息传递程序的基本问题，如数据的分割和分布式内存系统中的 I/O。我们还将重温关于并行程序性能的一些问题。

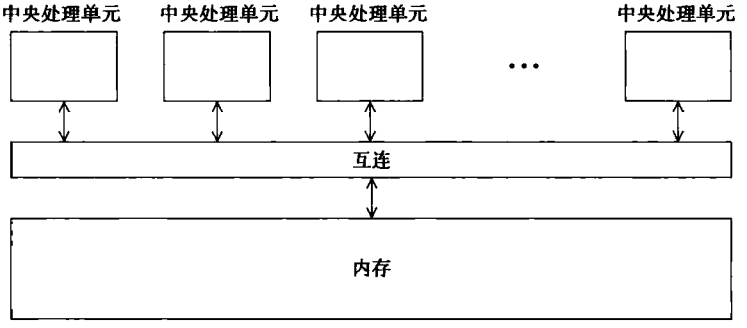



图 3-2　一个共享内存系统

## 3.1　预备知识

我们中大部分人的第一个程序可能都是从 Kernighan 和 Ritchie 的经典“hello，world”程序[29] 而来的：

```
#include <stdio.h>

int main(void){
   printf("hello, world n");

   return 0;
}
```

让我们用 MPI 来编写一个类似的“hello，world”程序。不让每个进程都简单地打印一条消息，相反，我们指派其中的一个进程负责输出，而其他进程向它发送要打印的消息。

在并行编程中，将进程按照非负整数来进行标注是非常常见的。如果有 $p$ 个进程，则这些进程将被编号为 0，1，2，…，$p-1$。对于我们的并行“hello，world”程序，我们指派 0 号进程为输出进程，其余进程向它发送消息。参见程序 3-1。

### 3.1.1　编译与执行

编译与运行程序的细节主要取决于系统，所以你可能需要询问本地专家。然而，当我们需要清晰地理解细节时，我们会假定使用一个文本编辑器来编写程序代码，并且用命令行形式来编译 [84] 和运行程序。许多系统都有称为 mpicc 的命令来编译程序[⊖]：

```
$ mpicc -g -Wall -o mpi_hello mpi_hello.c
```

典型情况下，mpicc 是 C 语言编译器的**包装脚本**（wrapper script）。包装脚本的主要目的是运行某个程序。在这种情况下，程序就是 C 语言编译器。通过告知编译器从何处取得需要的头文件、什么库函数连接到对象文件等，包装脚本可以简化编译器的运行。

**程序 3-1　打印来自进程问候语句的 MPI 程序**

```
 1  #include <stdio.h>
 2  #include <string.h>  /* For strlen             */
 3  #include <mpi.h>     /* For MPI functions, etc */
 4
 5  const int MAX_STRING = 100;
 6
 7  int main(void) {
 8     char       greeting[MAX_STRING];
 9     int        comm_sz;  /* Number of processes */
10     int        my_rank;  /* My process rank     */
11
12     MPI_Init(NULL, NULL);
13     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16     if (my_rank != 0) {
17        sprintf(greeting, "Greetings from process %d of %d!",
18              my_rank, comm_sz);
19        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20              MPI_COMM_WORLD);
21     } else {
22        printf("Greetings from process %d of %d!\n", my_rank,
            comm_sz);
23        for (int q = 1; q < comm_sz; q++) {
24           MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25              0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26           printf("%s\n", greeting);
27        }
28     }
29
30     MPI_Finalize();
31     return 0;
32  }  /* main */
```

⊖ 美元符号 $ 是 shell 提示符，它不需要输入。为保证表达的清晰，我们假定使用的是 Gnu C 编译器，gcc，我们一般总会使用 -g、-Wall 和 -o 选项。更多信息参见 2.9 节。

85 许多系统还支持用 mpiexec 命令来启动程序：

```
$ mpiexec -n <number of processes> ./mpi_hello
```

所以，为了用 1 个进程运行程序，我们输入，

```
$ mpiexec -n 1 ./mpi_hello
```

所以，为了用 4 个进程运行程序，我们输入，

```
$ mpiexec -n 4 ./mpi_nello
```

对于 1 个进程的程序，输出为，

```
Greetings from process 0 of 1!
```

对于 4 个进程的程序，输出为，

```
Greetings from process 0 of 4!
Greetings from process 1 of 4!
Greetings from process 2 of 4!
Greetings from process 3 of 4!
```

如何得到调用 mpiexec 命令所产生的结果呢？mpiexec 命令告诉系统启动 < number of processes > 个 < mpi_hello > 程序的实例。它或许还会告诉系统每个实例在哪个核上运行。当进程运行后，MPI 保证进程间可以互相通信。

### 3.1.2 MPI 程序

我们进一步观察这个程序。首先，这是一个 C 语言程序，它包含了 C 语言的标准头文件 stdio.h 和 string.h，它还有一个像其他 C 语言程序一样的 main 函数。然而，程序中的许多部分却是新的。第 3 行包含了 mpi.h 头文件。头文件包括了 MPI 函数的原形、宏定义、类型定义等，它还包括了编译 MPI 程序所需要的全部定义与声明。

我们观察到的第二件事是，所有 MPI 定义的标识符都由字符串 MPI_开始。下划线后的第一个字母大写，表示函数名和 MPI 定义的类型。MPI 定义的宏和常量的所有字母都是大写的，这样可以区分什么是 MPI 定义的，什么是用户程序定义的。

### 3.1.3 MPI_Init 和 MPI_Finalize

第 12 行中，调用 MPI_Init 是为了告知 MPI 系统进行所有必要的初始化设置。例如，系统可能需要为消息缓冲区分配存储空间，为进程指定进程号等。从经验上看，在程序调用 MPI_
86 Init 前，不应该调用其他 MPI 函数。它的语法结构为：

```
int MPI_Init(
      int*     argc_p  /* in/out */,
      char***  argv_p  /* in/out */);
```

参数 argc_p 和 argv_p 是指向参数 argc 和 argv 的指针。然而，当程序不使用这些参数时，可以只是将它们设置为 NULL。就像大部分的 MPI 函数一样，MPI_Init 返回一个 int 型错误码，在大部分情况下，我们忽略这些错误码。

第 30 行中，调用 MPI_Finalize 是为了告知 MPI 系统 MPI 已经使用完毕，为 MPI 而分配的任何资源都可以释放了。它的语法结构很简单：

```
int MPI_Finalize(void);
```

一般而言，在调用 MPI_Finalize 后，就不应该调用 MPI 函数了。

因此，一个典型的 MPI 程序有如下基本框架：

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

而且，我们已经知道，我们并不一定要向 MPI_Init 传递 agrc 和 argv 的指针，也不一定要在 main 函数中调用 MPI_Init 和 MPI_Finalize。

### 3.1.4 通信子、MPI_Comm_size 和 MPI_Comm_rank

在 MPI 中，**通信子**（communicator）指的是一组可以互相发送消息的进程集合。MPI_Init 的其中一个目的，是在用户启动程序时，定义由用户启动的所有进程所组成的通信子。这个通信子称为 MPI_COMM_WORLD。在第 13 行、第 14 行调用的函数可以获取关于 MPI_COMM_WORLD 的信息。这些函数的语法结构为：

```
int MPI_Comm_size(
      MPI_Comm  comm         /* in  */,
      int*      comm_sz_p    /* out */);

int MPI_Comm_rank(
      MPI_Comm  comm         /* in  */,
      int*      my_rank_p    /* out */);
```

这两个函数中，第一个参数是一个通信子，它所属的类型是 MPI 为通信子定义的特殊类型：MPI_Comm。MPI_Comm_size 函数在它的第二个参数里返回通信子的进程数；MPI_Comm_rank 函数在它的第二个参数里返回正在调用进程在通信子中的进程号。在 MPI_COMM_WORLD 中经常用参数 comm_sz 表示进程的数量，用参数 my_rank 来表示进程号。

### 3.1.5 SPMD 程序

注意，我们刚才编译的是单个程序，而不是为每个进程编译不同的程序。尽管 0 号进程本质上做的事情与其他进程不同。0 号进程所做的事，主要是当其他进程生成和发送消息时，它负责接收消息并打印出来。这在并行编程中很常见。事实上，大部分 MPI 程序都是这么写的。也就是说，编写一个单个程序，让不同进程产生不同动作。实现方式是，简单地让进程按照它们的进程号来匹配程序分支。这一方法称为单程序多数据流（Single Program，Multiple Data，SPMD）。第 16 行～第 28 行的 if-else 语句使得我们的程序是 SPMD 的。

另外需要注意的是，我们的程序原则上可以运行任意数量的进程。我们先前看到它可以在 1 个或 4 个进程上运行。但如果系统有足够的资源，那么可以在 1000 个，甚至 10 万个进程上运行。虽然 MPI 并不需要程序具有这样的属性，但是我们总是一直试着编写程序，使之可以运行任意数量的进程。这是因为我们通常事先并不知道可用的资源到底有多少。例如，现在有一个 20 核的系统，但是将来可能会拥有一个 500 核的系统。

### 3.1.6 通信

在第 17 行和第 18 行中，除了 0 号进程外，每个进程都生成了一条要发送给 0 号进程的消息（sprintf 函数和 printf 函数类似，它不是输出到标准输出 stdout，而是输出到一个字符串中）。第 19 行和第 20 行代码将消息发送给 0 号进程。另一方面，0 号进程只是用 printf 函数简

单地将消息打印出来，然后用一个 for 循环接收并打印由 1、2、…、comm_sz -1 号进程发送来的消息。第 24 行和第 25 行接收由 $q$ 号进程发送来的消息，其中，$q=1$、2、…、comm_sz -1。

### 3.1.7 MPI_Send

由 1、2、…、comm_sz -1 号进程执行的发送其实是很复杂的，我们可以进一步地看看里面是如何实现的。每个发送都是由调用 MPI_Send 来实现的，其语法结构为：

87
88

```
int MPI_Send(
      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```

前三个参数，msg_buf_p、msg_size 和 msg_type 定义了消息的内容。剩下的参数，dest、tag 和 communicator 定义了消息的目的地。

第一个参数 msg_buf_p 是一个指向包含消息内容的内存块的指针。在我们的程序中，这是一个包含了消息的字符串 greeting（在 C 语言中，像这样的字符串数组，其实是个指针）。第二个和第三个参数，msg_size 和 msg_type，指定了要发送的数据量。在我们的程序中，参数 msg_size 是消息字符串加上 C 语言中字符串结束符'\0'所占的字符数量。参数 msg_type 的值是 MPI_CHAR。这两个参数一起告知系统整个消息含有 strlen(greeting) +1 个字符。

因为 C 语言中的类型（int、char 等）不能作为参数传递给函数，所以 MPI 定义了一个特殊类型：MPI_Datatype，用于参数 msg_type。MPI 也为这个类型定义了一些常量，表 3-1 列出了要用到的一些数据类型。

**表 3-1 一些预先定义的 MPI 数据类型**

| MPI 数据类型 | C 语言数据类型 | MPI 数据类型 | C 语言数据类型 |
|---|---|---|---|
| MPI_CHAR | signed char | MPI_UNSIGNED | unsigned int |
| MPI_SHORT | signed short int | MPI_UNSIGNED_LONG | unsigned long int |
| MPI_INT | signed int | MPI_FLOAT | float |
| MPI_LONG | signed long int | MPI_DOUBLE | double |
| MPI_LONG_LONG | signed long long int | MPI_LONG_DOUBLE | long double |
| MPI_UNSIGNED_CHAR | unsigned char | MPI_BYTE | |
| MPI_UNSIGNED_SHORT | unsigned short int | MPI_PACKED | |

我们注意到，字符串 greeting 的大小与 msg_size 和 msg_type 所指定的消息的大小并不
89 相同。例如，当我们用 4 个进程运行程序时，每条消息的长度为 31 个字符，但我们分配长度为 100 个字符的缓冲区来存储 greeting 字符串。当然，发送消息的大小必须小于或等于缓冲区的大小，如程序中的 greeting 字符串。

第四个参数 dest 指定了要接收消息的进程的进程号。第五个参数 tag 是个非负 int 型，用于区分看上去完全一样的消息。例如，假定 1 号进程正在向 0 号进程发送浮点数，其中一些浮点数需要打印出来，而另一些用于计算。那么，MPI_Send 的前 4 个参数并不能提供相应的信息，从而知道哪些浮点数是需要打印的，哪些又是用于计算的。所以，1 号进程可以用，也就是说，tag 标签为 0 就表示消息是要打印的，为 1 就表示消息是用于计算的。

MPI_Send 的最后一个参数是一个通信子。所有涉及通信的 MPI 函数都有一个通信子参数。

通信子最重要的目的之一是指定通信范围。通信子指的是一组可以互相发送消息的进程的集合。反过来，一个通信子中的进程所发送的消息不能被另一个通信子中的进程所接收。由于MPI提供了创建新通信子的函数，因此通信子这一特性可以用于复杂程序，并保证消息不会意外地在错误的地方被接收。

举个例子来阐明这一特性。假设我们正在研究全球气候变化，并且很幸运地得到了两个函数库，一个用来对地球大气层进行建模，另一个用来对地球上的海洋进行建模。当然，这两个库都使用了MPI。这些模型是分开建立的，所以它们之间不会相互通信，但它们内部可以进行通信。我们的工作就是编写接口代码。我们需要解决的一个问题是，要保证一个库发送的消息不会意外地被另一个库所接收。我们可能会利用标签来制定解决方案：大气层函数库使用标签0、1、…、$n-1$；海洋函数库使用标签$n$、$n+1$、…、$n+m$。那么每个函数库就可以使用给定的范围给出消息的标签。然而，一个更为简单的方法是由通信子提供的，我们只需要简单地给大气层库函数一个通信子，将另一个通信子给海洋库函数就行了。

### 3.1.8 MPI_Recv

MPI_Recv的前六个参数对应了MPI_Send的前六个参数：

```
int MPI_Recv(
      void*          msg_buf_p     /* out */,
      int            buf_size      /* in  */,
      MPI_Datatype   buf_type      /* in  */,
      int            source        /* in  */,
      int            tag           /* in  */,
      MPI_Comm       communicator  /* in  */,
      MPI_Status*    status_p      /* out */);
```

因此，前三个参数指定了用于接收消息的内存：msg_buf_p指向内存块，buf_size指定了内存块中要存储对象的数量，buf_type说明了对象的类型。后面的三个参数用来识别消息。参数source指定了接收的消息应该从哪个进程发送而来，参数tag要与发送消息的参数tag相匹配，参数communicator必须与发送进程所用的通信子相匹配。简短地介绍参数status_p。在大部分情况下，调用函数并不使用这个参数，就像我们的“greeting”程序中那样，赋予其特殊的MPI常量MPI_STATUS_IGNORE就行了。 [90]

### 3.1.9 消息匹配

假定$q$号进程调用了MPI_Send函数：

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

并且假定$r$号进程调用了MPI_Recv函数：

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

则$q$号进程调用MPI_Send函数所发送的消息可以被$r$号进程调用MPI_Recv函数接收，如果：

- recv_comm = send_comm,
- recv_tag = send_tag,
- dest = r,并且
- src = q.

然而，这些条件还不足以使消息可以成功地接收。前三对参数：send_buf_p / recv_buf_p、send_buf_sz / recv_buf_sz和send_type / recv_type必须指定兼容的缓冲区。详细的规则，请参见MPI-1说明[39]。大多数时候，满足下面的规则就可以了：

- 如果 recv_type = send_type，同时 recv_buf_sz ≥ send_buf_sz，那么由 $q$ 号进程发送的消息就可以被 $r$ 号进程成功地接收。

当然，一个进程可以接收多个进程发来的消息，接收进程并不知道其他进程发送消息的顺序。例如，假设0号进程将任务分发给1、2、…、comm_sz－1号进程，并且1、2、…、comm_sz－1号进程在完成工作时将结果发还给0号进程。如果分配给每个进程的工作所要耗费的时间是无法预测的，那么0号进程就无法知道其他进程完成工作的顺序。如果0号进程只是简单地按照进程号顺序地接收结果，即先接收1号进程的结果，再接收2号进程的结果，以此类推，并且，如果 commsz－1号进程是第一个完成工作的，那么有可能 comm_sz－1号必须等待其他进
91 程的完成。为了避免这个问题，MPI 提供了一个特殊的常量 MPI_ANY_SOURCE，可以传递给 MPI_Recv。这样，如果0号进程执行下列代码，那么它可以按照进程完成工作的顺序来接收结果了：

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
         result_tag, comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

类似地，一个进程也有可能接收多条来自另一个进程的有着不同标签的消息，并且接收进程并不知道消息发送的顺序。在这种情况下，MPI 提供了特殊常量 MPI_ANY_TAG，可以将它传给 MPI_Recv 的参数 tag。

当使用这些“通配符”（wildcard）参数时，有几点需要强调：

1）只有接收者可以使用通配符参数。发送者必须指定一个进程号与一个非负整数标签。因此，MPI 使用的是所谓的“推”（push）通信机制，而不是“拉”（pull）通信机制。

2）通信子参数没有通配符。发送者和接收者都必须指定通信子。

### 3.1.10 status_p 参数

如果你仔细想想我们所说的这些规则，你会发现接收者可以在不知道以下信息的情况下接收消息：

1）消息中的数据量，

2）消息的发送者，或

3）消息的标签。

那么，接收者是如何找出这些值的？回想一下，MPI_Recv 的最后一个参数的类型为 MPI_Status *。MPI 类型 MPI_Status 是一个有至少三个成员的结构，MPI_SOURCE、MPI_TAG 和 MPI_ERROR。假定程序含有如下的定义：

```
MPI_Status status;
```

那么，将 &status 作为最后 -个参数传递给 MPI_Recv 函数并调用它后，可以通过检查以下两个成员来确定发送者和标签：

```
status.MPI_SOURCE
status.MPI_TAG
```

接收到的数据量不是存储在应用程序可以直接访问到的域中，但用户可以调用 MPI_Get_count 函数找回这个值。例如，假设对 MPI_Recv 的调用中，接收缓冲区的类型为 recv_type，再次传递 &status 参数，则以下调用

```
MPI_Get_count(&status, recv_type, &count)
```

92 会返回 count 参数接收到的元素数量。一般而言，MPI_Get_count 的语法结构为：

```
int MPI_Get_count(
        MPI_Status*  status_p  /* in  */,
        MPI_Datatype type      /* in  */,
        int*         count_p   /* out */);
```

注意，count 值并不能简单地作为 MPI_Status 变量的成员直接访问，因为它取决于接收数据的类型。因此，确定该值的过程需要一次计算（如接收到的字节数 / 每个对象的字节数）。如果这个信息不是必须的，那么我们没必要为了得到该值浪费一次计算。

### 3.1.11 MPI_Send 和 MPI_Recv 的语义

当我们将消息从一个进程发送到另一个进程时，会发生什么？这其中的许多细节取决于具体的系统，但我们可以有一些一般化的概念。发送进程组装消息，例如，它为实际要发送的数据添加“信封”信息。“信封”信息包括目标进程的进程号、发送进程的进程号、标签、通信子，以及消息大小等信息。一旦消息组装完毕，如第 2 章所说，有两种可能性：发送进程可以**缓冲**消息，也可以**阻塞**（block）。如果它缓冲消息，则 MPI 系统将会把消息（包括数据和信封）放置在它自己的内部存储器里，并返回 MPI_Send 的调用。

另一方面，如果系统发生阻塞，那么它将一直等待，直到可以开始发送消息，并不立即返回对 MPI_Send 的调用。因此，如果使用 MPI_Send，当函数返回时，实际上并不知道消息是否已经发送出去。我们只知道消息所用的存储区，即发送缓冲区，可以被程序再次使用。如果我们需要知道消息是否已经发送出去，或者无论消息是否已经发送出去我们都让 MPI_Send 调用后立即返回，那么可以使用 MPI 提供的发送消息的替代方法。我们将会在稍后学习这些替代函数中的一个。

MPI_Send 的精确行为是由 MPI 实现所决定的。但是，典型的实现方法有一个默认的消息“截止”大小（“cutoff” message size）。如果一条消息的大小小于“截止”大小，它将被缓冲；如果大于截止大小，那么 MPI_ Send 函数将被阻塞。

与 MPI_Send 不同，MPI_Recv 函数总是阻塞的，直到接收到一条匹配的消息。因此，当 MPI_Recv 函数调用返回时，就知道一条消息已经存储在接收缓冲区中了（除非产生了错误）。接收消息函数同样有替代函数，系统检查是否有一条匹配的消息并返回，而不管缓冲区中有没有消息（关于使用非阻塞通信的细节，详见习题 6.22）。

MPI 要求消息是**不可超越的**（nonovertaking）。即如果 $q$ 号进程发送了两条消息给 $r$ 号进程，那么 $q$ 进程发送的第一条消息必须在第二条消息之前可用。但是，如果消息是来自不同进程的，[93] 消息的到达顺序是没有限制的。即如果 $q$ 号进程和 $t$ 号进程都向 $r$ 号进程发送了消息，即使 $q$ 号进程在 $t$ 号进程发送消息之前就将自己的消息发送出去了，也不要求 $q$ 号进程的消息在 $t$ 号进程的消息之前一定能被 $r$ 号进程所访问。这本质上是因为 MPI 不能对网络的性能有强制性要求。例如，如果 $q$ 号进程在火星上的某台机器上运行，而 $r$ 号进程和 $t$ 号进程都在旧金山的同一台机器上运行，并且 $q$ 号进程只是在 $t$ 号进程发送消息之前的 1 纳秒发送了消息，那么要求 $q$ 号进程的消息在 $t$ 号进程之前到达，是不合理的。

### 3.1.12 潜在的陷阱

MPI_Recv 的语义会导致 MPI 编程中的一个潜在陷阱：如果一个进程试图接收消息，但没有相匹配的消息，那么该进程将会被永远阻塞在那里，即进程**悬挂**。因此，在设计程序时，我们需要保证每条接收都有一条相匹配的发送。可能更重要的是，编写代码时，要格外小心以防止因调用 MPI_Send 和 MPI_Recv 出现错误。例如，如果标签（tag）不匹配，或者目标进程的进程号与源进程的进程号不相同，那么接收与发送就无法相匹配了，这会导致一个进程悬挂起来，或者可能更严重的，接收端可能会匹配另一个发送端。

简单地说，如果调用 MPI_Send 发生了阻塞，并且没有相匹配的接收，那么发送进程就悬挂起来。另一方面，如果调用 MPI_ Send 被缓冲，但没有相匹配的接收，那么消息将被丢失。

## 3.2 用 MPI 来实现梯形积分法

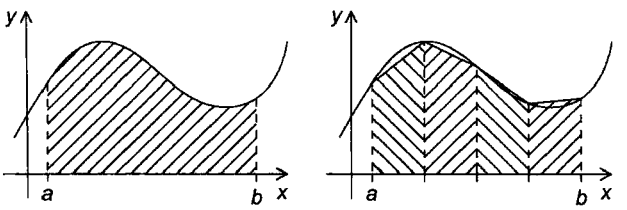
编写打印来自进程消息的程序比较简单，一般不会遇到什么麻烦，程序也会运行良好。但是，我们不会只编写打印消息的程序。接下来是一个更为有用的程序：通过编写程序来实现数值积分中的梯形积分法。

### 3.2.1 梯形积分法

我们可以用梯形积分法来估计函数 $y=f(x)$ 的图像中，两条垂直线与 $x$ 轴之间的区域大小，见图 3-3。基本思想是，将 $x$ 轴上的区间划分为 $n$ 个等长的子区间。然后估计介于函数图像及每个子区间内的梯形区域的面积。梯形的底边是 $x$ 轴上的子区间，两条垂直边是经过子区间端点的垂直线，第四条边是两条垂直边与函数图像所相交的两个交点之间的连线。如图 3-4 所示，设子区间的端点为 $x_i$ 和 $x_{i+1}$，那么子区间的长度 $h=x_{i+1}-x_i$。同样，两条垂直线段的长度分别为 $f(x_i)$ 和 $f(x_{i+1})$，那么该梯形区域的面积就为：

94

$$\text{梯形面积}=\frac{h}{2}[f(x_i)+f(x_{i+1})]$$




a) b)

图 3-3 梯形积分法：a）要估计的区域；b）用梯形近似的区域

由于 $n$ 个子区间是等分的，因此如果两条垂直线包围区域的边界分别为 $x=a$ 和 $x=b$，那么

$$h=\frac{b-a}{n}$$

因此，如果称最左边的端点为 $x_0$，最右边的端点为 $x_n$，则有：

$$x_0=a, x_1=a+h, x_2=a+2h, \cdots, x_{n-1}=a+(n-1)h, x_n=b$$

这片区域的所有梯形的面积和为

$$\text{梯形面积和}=h[f(x_0)/2+f(x_1)+\cdots+f(x_{n-1})+f(x_n)/2]$$

因此，一个串行程序的伪代码有可能看起来是这样的：

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++){
   x_i = a + i*h;
   approx += f(x_i);
}
```

95

```
approx = h*approx;
```

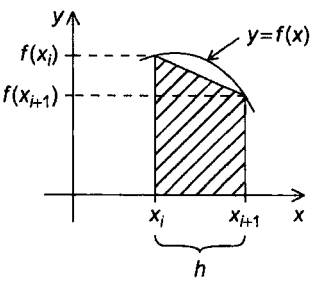



图 3-4 一个梯形

### 3.2.2 并行化梯形积分法

正如第 1 章所言，编写并行程序的程序员通常用“并行化”来描述将串行程序或算法转换为并行程序的过程。

回想一下，可以用四个基本步骤去设计一个并行程序：

1）将问题的解决方案划分成多个任务。

2）在任务间识别出需要的通信信道。

3）将任务聚合成复合任务。

4）在核上分配复合任务。

在划分阶段，我们通常试着识别出尽可能多的任务。对于梯形积分法，我们可以识别出两种任务：一种是获取单个矩形区域的面积，另一种是计算这些区域的面积和。然后利用通信信道将每个第一种任务与一个第二种任务相连接，见图 3-5。

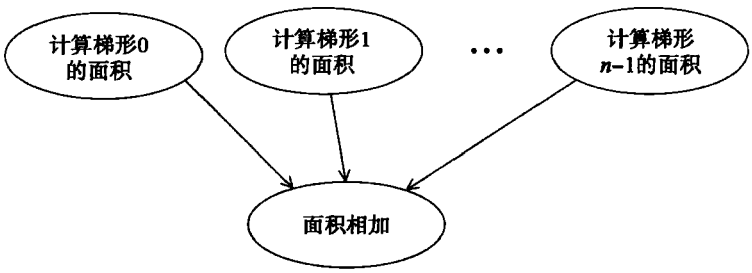



图 3-5　梯形积分法的任务与通信

那么，如何聚合任务并将其分配到核上呢？直觉告诉我们，使用的梯形越多，估计值就越精确。也就是说：应该使用尽可能多的梯形。因此，梯形的数目将超过核的数量，需要将梯形区域面积的计算聚合成组。实现这一目标的一个很自然的方法就是将区间 $[a, b]$ 分成 comm_sz 个子区间。如果 comm_sz 可以整除 $n$，即梯形数目，那么我们可以简单地在 $n$/comm_sz 个梯形和所有 comm_sz 个子空间上应用梯形积分法。最后，我们可以利用进程中的某一个，如 0 号进程，将这些梯形面积的估计值累加起来，完成整个计算过程。

我们简单地假设 comm_sz 能整除 $n$，则这个程序的伪代码如下所示。

```
1  Get a, b, n;                                                    96
2  h = (b-a)/n;
3  local_n = n/comm_sz;
4  local_a = a + my_rank*local_n*h;
5  local_b = local_a + local_n*h;
6  local_integral = Trap(local_a, local_b, local_n, h);
7  if (my_rank != 0)
8     Send local_integral to process 0;
9  else /* my_rank == 0 */
10    total_integral = local_integral;
11    for (proc = 1; proc < comm_sz; proc++) {
12       Receive local_integral from proc;
13       total_integral += local_integral;
14    }
15 }
16 if (my_rank == 0)
17    print result;
```

我们先暂时推迟考虑输入的问题，而只是将 $a$、$b$ 和 $n$ 视为常量。这样就会得到程序 3-2 所示的 MPI 程序。Trap 函数是一个梯形积分法的串行实现，见程序 3-3。

注意，我们对标识符的选择，是为了区分*局部*变量与*全局*变量。**局部**变量只在使用它们的进程内有效。梯形积分法程序中的例子有：local_a、local_b 和 local_n。如果变量在所有进程中都有效，那么该变量就称为**全局**变量。该程序中的例子有：变量 a、b 和 n。这与你在编程导论课上学到的用法不同。在编程导论课上，局部变量指的是单个函数的私有变量，而全局变量是指所有函数都可以访问的变量。但是，这不会引起混淆，因为只要通过上下文就可以理解到底

指的是哪种使用方法。

## 3.3 I/O 处理

现有版本的并行梯形积分法程序有个严重的不足：它只能用 1024 个梯形计算 [0, 3] 区间内的积分。与简单地输入三个新值相比，编辑和重新编译代码这种做法的工作量相当大。因此我们需要解决用户输入的问题。当讨论并行程序输入的同时，我们也一并将输出问题考虑进来。在第 2 章，我们讨论过这两个问题，所以如果你还记得对于不确定性和输出的讨论，那么你可以跳过本节，直接从 3.3.2 开始阅读。

### 3.3.1 输出

在“问候”程序和梯形积分法程序中，假定 0 号进程将结果写到标准输出 stdout，即它对 printf 函数的调用是我们所希望的行为。虽然 MPI 标准没有指定哪些进程可以访问哪些 I/O 设备，但是几乎所有的 MPI 实现都允许 MPI_COMM_WORLD 里的所有进程都能访问标准输出 stdout 和标准错误输出 stderr，所以，大部分的 MPI 实现都允许所有进程执行 printf 和 fprintf (stderr, …)。

但是，大部分的 MPI 实现并不提供对这些 I/O 设备访问的自动调度。也就是说，如果多个进
97 程试图写标准输出 stdout，那么这些进程的输出顺序是无法预测的，甚至会发生一个进程的输出被另一个进程的输出打断的情况。

**程序 3-2 梯形积分法 MPI 程序的第一个版本**

```
1  int main(void) {
2     int my_rank, comm_sz, n = 1024, local_n;
3     double a = 0.0, b = 3.0, h, local_a, local_b;
4     double local_int, total_int;
5     int source;
6
7     MPI_Init(NULL, NULL);
8     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11    h = (b-a)/n;           /* h is the same for all processes */
12    local_n = n/comm_sz;   /* So is the number of trapezoids  */
13
14    local_a = a + my_rank*local_n*h;
15    local_b = local_a + local_n*h;
16    local_int = Trap(local_a, local_b, local_n, h);
17
18    if (my_rank != 0) {
19       MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       total_int = local_int;
23       for (source = 1; source < comm_sz; source++) {
24          MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          total_int += local_int;
27       }
28    }
29
30    if (my_rank == 0) {
31       printf("With n = %d trapezoids, our estimate\n", n);
32       printf("of the integral from %f to %f = %.15e\n",
33           a, b, total_int);
34    }
35    MPI_Finalize();
36    return 0;
37 } /* main */
```

例如，假设运行一个 MPI 程序，每个进程只是简单地打印一条消息，见程序 3-4。在我们的集群上，如果用 5 个进程来执行这一程序，通常它会产生如下希望得到的输出：

98

```
Proc 0 of 5 > Does anyone have a toothpick?
Proc 1 of 5 > Does anyone have a toothpick?
Proc 2 of 5 > Does anyone have a toothpick?
Proc 3 of 5 > Does anyone have a toothpick?
Proc 4 of 5 > Does anyone have a toothpick?
```

**程序 3-3　梯形积分法 MPI 程序里的 Trap 函数**

```
1  double Trap(
2        double left_endpt  /* in */,
3        double right_endpt /* in */,
4        int    trap_count  /* in */,
5        double base_len    /* in */) {
6     double estimate, x;
7     int i;
8
9     estimate = (f(left_endpt) + f(right_endpt))/2.0;
10    for (i = 1; i <= trap_count-1; i++) {
11       x = left_endpt + i*base_len;
12       estimate += f(x);
13    }
14    estimate = estimate*base_len;
15
16    return estimate;
17 } /*  Trap  */
```

**程序 3-4　每个进程只是打印一条消息**

```
#include <stdio.h>
#include <mpi.h>

int main(void) {
   int my_rank, comm_sz;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Proc %d of %d > Does anyone have a toothpick?\n",
         my_rank, comm_sz);

   MPI_Finalize();
   return 0;
}  /* main */
```

但是，当用 6 个进程运行程序时，输出行的顺序就不可预测了：

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
```

99

或者：

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

这一现象产生的原因是 MPI 进程都在相互“竞争”，以取得对共享输出设备、标准输出 stdout 的访问。我们不可能预测进程的输出是以怎样的顺序排列。这种竞争会导致不确定性，即每次运行的实际输出可能会变化。

如果不希望进程的输出以随机顺序出现，那么我们就应该按自己的想法去修改代码。例如，让除了 0 号进程以外的其他进程向 0 号进程发送它的输出，然后 0 号进程根据进程号的顺序打印输出结果。这就是我们在“问候”程序中所做的。

### 3.3.2 输入

与输出不同，大部分的 MPI 实现只允许 MPI_COMM_WORLD 中的 0 号进程访问标准输入 stdin。这是有道理的：如果多个进程都能访问标准输入 stdin，那么哪个进程应该得到输入数据的哪个部分呢？0 号进程应该得到第一个字符吗？

为了编写能够使用 scanf 的 MPI 程序，我们根据进程号来选取转移分支。0 号进程负责读取数据，并将数据发送给其他进程。例如，在梯形积分法的并行程序（程序 3-5）中的 Get_input 函数，0 号进程只是简单地读取 $a$、$b$ 和 $n$ 的值，并将这三个值发送给其他每个进程。除了 0 号进程发送数据给其他进程、其他进程只是接收数据外，程序 3-5 使用了与“问候”程序相同的基本通信结构。

为了使用该函数，我们可以在主程序中简单地插入对该函数的调用。要注意的是，我们必须
100 在初始化 my_rank 和 comm_sz 后，才能调用该函数：

```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_data(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
. . .
```

**程序 3-5 一个用于读取用户输入的函数**

```
1  void Get_input(
2        int      my_rank   /* in  */,
3        int      comm_sz   /* in  */,
4        double*  a_p       /* out */,
5        double*  b_p       /* out */,
6        int*     n_p       /* out */) {
7     int dest;
8
9     if (my_rank == 0) {
10       printf("Enter a, b, and n\n");
11       scanf("%lf %lf %d", a_p, b_p, n_p);
12       for (dest = 1; dest < comm_sz; dest++) {
13          MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14          MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15          MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16       }
17    } else { /* my_rank != 0 */
18       MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
19             MPI_STATUS_IGNORE);
20       MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
21             MPI_STATUS_IGNORE);
22       MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
23             MPI_STATUS_IGNORE);
24    }
25 }  /* Get_input */
```

## 3.4　集合通信

如果停下来，想一想我们编写的梯形积分法程序，我们就会发现几件可以提高程序性能的事。其中最明显的就是在每个进程都完成了它那部分积分任务之后的“全局求和”（global sum）。想象一下，如果我们雇用8名工人建造一所房子，那么如果其中的7名工人告诉第一个人如何干，然后将他们的7份工钱拿走回家，我们会觉得所花的钱非常不值。但这很像全局求和所做的：每个进程号大于0的进程“告知0号进程怎么做”，然后就自己退出了。每个进程号大于0的进程事实上只是说“将这个值加到总和里去”。0号进程要做全局求和的所有工作，而其他进程几乎什么都不做。有时候，这可能是并行程序可以选择的最好方式，但如果想象一下有8名学生，每个人都拥有一个数值，为了求这8个数值的总和，相比起其中7名学生将自己的数值交给第一个学生，并让他来进行求和工作，我们可以想出一个更公平的工作分配方法。 101

### 3.4.1　树形结构通信

正如我们在第1章中见到的那样，可以使用一棵像图3-6所描绘的二叉树结构。在图3-6中，一开始1号进程、3号进程、5号进程、7号进程将它们的值分别发送给0号进程、2号进程、4号进程、6号进程。然后0号进程、2号进程、4号进程、6号进程将接收到的值加到它们自己原有的值上，整个过程重复两次：

1）a. 2号进程和6号进程将它们的新值分别发送给0号进程和4号进程。

b. 0号进程和4号进程将接收到的值加到它们的新值上。

2）a. 4号进程将它最新的值发送给0号进程。

b. 0号进程将接收到的值加到它最新的值上。

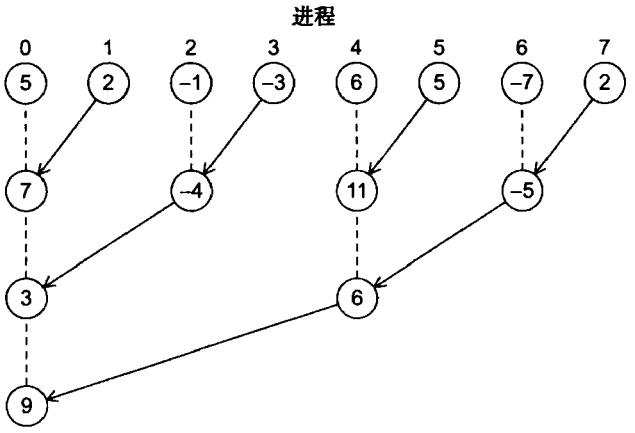



图3-6　树形结构全局求和

这个解决方案可能不是最理想的，因为其中一半进程（1号进程、3号进程、5号进程和7号进程）的工作量与原有方案相同。但是，如果我们仔细想想，就会发现原有方案需要0号进程接收 comm_sz - 1 = 7 次，并做7次加法。而新方案中0号进程只需要3次接收和3次加法，并且其他进程所做的接收与加法操作都不超过2次。此外，新方案拥有这样的特性，即许多工作是由不同进程并发完成的。例如，在第一个阶段里，0号进程、2号进程、4号进程、6号进程的接收与加法操作可以同时进行。所以，如果进程几乎是同时启动的，那么全局求和需要的总时间将是0号进程需要的时间，即3次接收操作与3次加法操作。因此减少了超过50%的总时间。另外，如果使用更多的进程，甚至可以得到更好的结果。例如，如果 comm_sz = 1024，则原方案需要0号进程执行1023次接收及加法操作，而采用新方案，0号进程只需要10次接收及加法操作（见习题3.5）。这使得原方案的性能提高了超过100倍！ 102

你会觉得这种做法相当好，但是编写树形结构代码看上去需要做许多工作，参见编程作业3.3。事实上，这个问题甚至可能更加困难。例如，建立一个基于树形结构的全局求和，使之用于不同的“进程配对”，这种做法是完全可行的。例如，在第一阶段中，可以将0号进程与4号进程配对，1号进程与5号进程相配，2号进程与6号进程相配，3号进程与7号进程相配。在第二阶段，将0号进程与2号进程配对，1号进程与3号进程配对。最后，将0号进程与1号进程配

对。参见图 3-7。当然，还有许多其他可能的配对法。我们能够确定哪个方案是最优的吗？如果可以，会不会有这种可能，即一种方案对于“小”的树形结构是最优的，而另一种方案对于“大”的树形结构是最优的？更坏情况是：一种方案可能在系统 A 上是最优的，而另一种方案在系统 B 上是最优的。

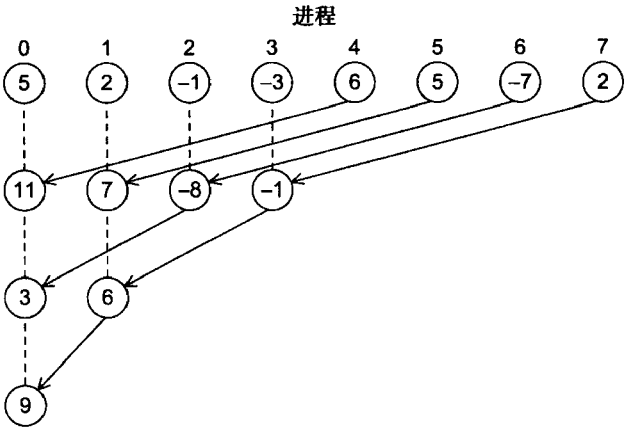



图 3-7　树形结构全局求和的另一种方法

### 3.4.2　MPI_Reduce

由于存在各种可能性，指望 MPI 程序员都能编写出最佳的全局求和函数是不合理的。所以，MPI 中包含了全局求和的实现，以便帮助程序员摆脱无止境的程序优化。这样，就将程序优化的压力转移到 MPI 实现的开发人员身上，而不在应用程序的开发人员身上。假设 MPI 实现的开发人员对硬件和系统软件都应该有足够的了解，这样才能更好地确定实现细节。

很明显，“全局求和函数”需要通信。然而，与 MPI_Send 函数和 MPI_Recv 函数两两配对不同，全局求和函数可能会涉及两个以上的进程。事实上，在梯形积分法程序中，它涉及 MPI_COMM_WORLD 中所有的进程。在 MPI 里，涉及通信子中所有进程的通信函数称为**集合通信**（collective communication）。为了区分集合通信与类似 MPI_Send 和 MPI_Recv 这样的函数，MPI_Send 和 MPI_Recv 通常称为**点对点通信**（point-to-point communication）。

103

实际上，全局求和函数只是集合通信函数类别中的一个特殊例子而已。例如，我们可能并不是想计算分布在各个进程上的 comm_sz 值的总和，而是想知道最大值或最小值，或者总的乘积，或者其他许多可能情况中的任何一个所产生的结果。MPI 对全局求和函数进行概括，使这些可能性中的任意一个都能用单个函数来实现：

```
int MPI_Reduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      int           dest_process   /* in  */,
      MPI_Comm      comm           /* in  */);
```

这个函数的关键在于第 5 个参数，operator。它的类型为 MPI_Op，是一个像 MPI_Datatype 和 MPI_Comm 一样的预定义 MPI 类型。这个类型有多个预定义值，见表 3-2。你还可以定义自己的运算符，详见 MPI-1 标准 [39]。

这里，我们要使用的运算符为 MPI_SUM，将这个值赋给 operator 参数后，就可以将程序 3-2 中的第 18 ~ 28 行的代码用单个函数调用代替：

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

需要注意的是，如果 count 参数大于 1，那么 MPI_Reduce 函数可以应用到数组上，而不仅仅是应用在简单的标量上。下面的代码可以用于一组 $N$ 维向量的加法，每个进程上有一个向量：

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

表3-2　MPI中预定义的归约操作符

| 运算符值 | 含　义 | 运算符值 | 含　义 |
|---|---|---|---|
| MPI_MAX | 求最大值 | MPI_LOR | 逻辑或 |
| MPI_MIN | 求最小值 | MPI_BOR | 按位或 |
| MPI_SUM | 求累加和 | MPI_LXOR | 逻辑异或 |
| MPI_PROD | 求累乘积 | MPI_BXOR | 按位异或 |
| MPI_ LAND | 逻辑与 | MPI_MAXLOC | 求最大值和最大值所在的位置 |
| MPI_BAND | 按位与 | MPI_MINLOC | 求最小值和最小值所在的位置 |

### 3.4.3　集合通信与点对点通信

记住，集合通信与点对点通信在多个方面是不同的：

1）在通信子中的所有进程都必须调用相同的集合通信函数。例如，试图将一个进程中的MPI_Reduce调用与另一个进程的MPI_Recv调用相匹配的程序会出错，此时程序会被悬挂或者崩溃。

2）每个进程传递给MPI集合通信函数的参数必须是“相容的”。例如，如果一个进程将0作为dest_process的值传递给函数，而另一个传递的是1，那么对MPI_Reduce调用所产生的结果就是错误的，程序可能被悬挂起来或者崩溃。

3）参数output_data_p只用在dest_process上。然而，所有进程仍需要传递一个与output_data_p相对应的实际参数，即使它的值只是NULL。

4）点对点通信函数是通过标签和通信子来匹配的。集合通信函数不使用标签，只通过通信子和调用的顺序来进行匹配。例如，看看表3-3所示的MPI_Reduce调用，假设每个进程调用MPI_Reduce函数的运算符都是MPI_SUM，那么目标进程为0号进程。粗略地看一下整张表，在两次调用MPI_Reduce后，b的值是3，而d的值是6。但是，内存单元的名字与MPI_Reduce的调用匹配无关，函数调用的顺序决定了匹配方式。所以b中所存储的值将是1+2+1=4，d中存储的值将是2+1+2=5。

表3-3　对MPI_Reduce的多个调用

| 时间 | 0号进程 | 1号进程 | 2号进程 |
|---|---|---|---|
| 0 | a = 1; c = 2 | a = 1; c = 2 | a = 1; c = 2 |
| 1 | MPI_Rdeuce (&a,&b,... ) | MPI_Rdeuce (&c,&d,... ) | MPI_Rdeuce (&a,&b,... ) |
| 2 | MPI_Rdeuce (&c,&d,... ) | MPI_Rdeuce (&a,&b,... ) | MPI_Rdeuce (&c,&d,... ) |

最后一个忠告：我们也许会冒风险使用同一个缓冲区同时作为输入和输出调用MPI_Reduce。例如，我们想求得所有进程里x的全局总和，并且将x的结果放在0号进程里，也许会试着这样调用：

```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

104
105

但在MPI里，这种调用方式是非法的。它的结果是不可预测的：它可能产生一个错误结果，也可能导致程序崩溃，但也可能产生正确的结果。它之所以是非法的，主要原因是它涉及输出参数的别名。两个参数如果指向的是同一块内存，它们之间就存在别名问题。MPI禁止输入或输出参数作为其他参数的别名。因为MPI论坛希望使Fortran语言与C语言版本的MPI尽可能一致，Fortran语言禁止使用别名参数。在某些例子中，MPI提供了一种替代的构造方法，可以有效地避免这一

限制，见6.1.9节中的例子。

### 3.4.4 MPI_Allreduce

梯形积分法程序中，我们只打印结果，所以只用一个进程来得到全局总和的结果是很自然的。然而，不难想象这样一个情况，即所有进程都想得到全局总和的结果，以便可以完成一个更大规模的计算。在这个情况下，遇到的问题与刚才遇到的问题其实是相同的。例如，用一棵树来计算全局总和，我们可以通过“颠倒”（reverse）整棵树来发布全局总和（见图3-8）。还有另一种替代方法，可以让进程之间相互交换部分结果，而不是单向的通信。这种通信模式称为**蝶形**（见图3-9）。使用哪个通信结构，以及如何编写代码优化代码性能，都是程序员不希望完成的工作。幸运的是，MPI提供了一个MPI_Reduce的变种，可以令通信子中的所有进程都存储结果：

```
int MPI_Allreduce(
        void*         input_data_p   /* in  */,
        void*         output_data_p  /* out */,
        int           count          /* in  */,
        MPI_Datatype  datatype       /* in  */,
        MPI_Op        operator       /* in  */,
        MPI_Comm      comm           /* in  */);
```

参数表其实与MPI_Reduce的是相同的，除了没有dest_process这个参数，因为所有进程都能得到结果。

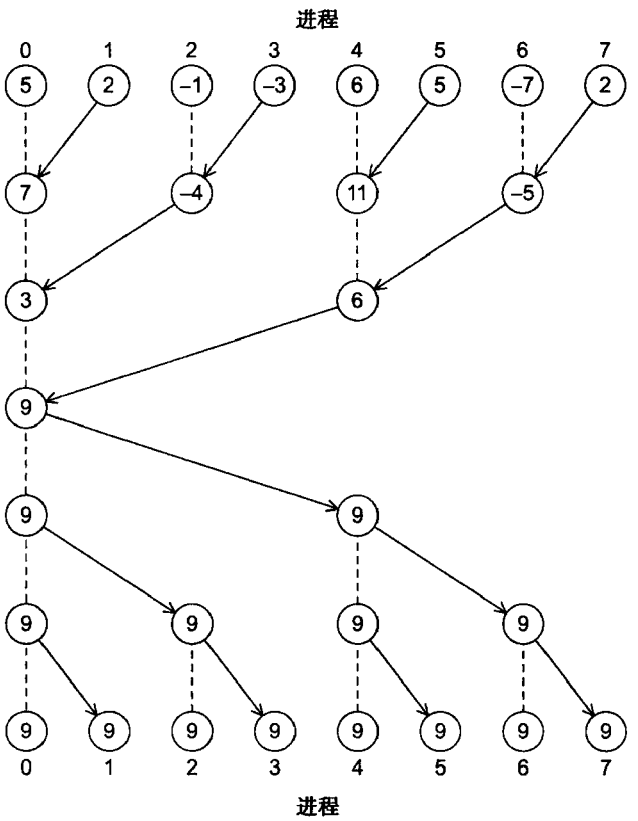



图3-8 全局求和计算结果的发布

### 3.4.5 广播

在梯形积分法程序中，如果用树形结构的通信来代替0号进程的循环接收消息，从而提升全

局求和的性能，那么我们应该也可以将类似的方法用于输入数据的发布。事实上，如果简单地在树形全局求和里“颠倒”通信，如图 3-6 所示，我们可以得到如图 3-10 所示的树形结构通信图，并且能够将这个结构用于输入数据的发布。在一个集合通信中，如果属于一个进程的数据被发送到通信子中的所有进程，这样的集合通信就叫做**广播**（broadcast）。你可能已经猜到了，MPI 提供 106 这样的一个广播函数：

```
int MPI_Bcast(
      void*         data_p          /* in/out */,
      int           count           /* in     */,
      MPI_Datatype  datatype        /* in     */,
      int           source_proc     /* in     */,
      MPI_Comm      comm            /* in     */);
```

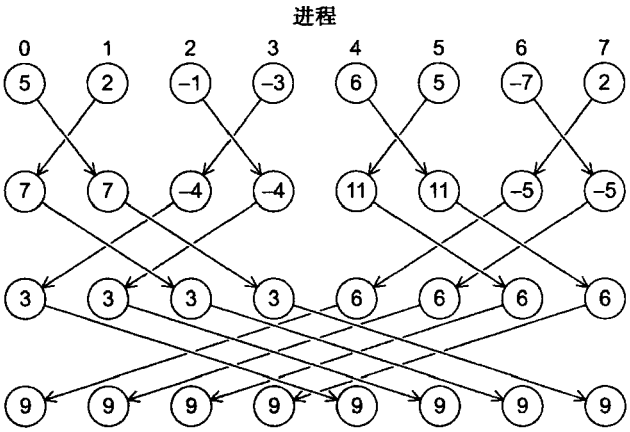



图 3-9　蝶形结构的全局求和

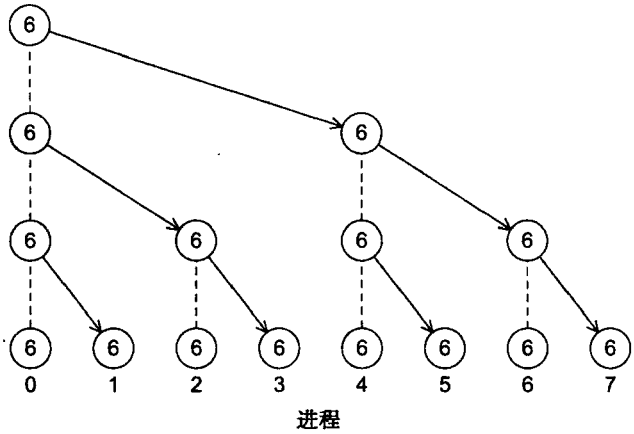



图 3-10　树形结构的广播

进程号为 source_proc 的进程将 data_p 所引用的内存内容发送给了通信子 comm 中的所有进程。程序 3-6 说明如何修改程序 3-5 中的 Get_input 函数，使它可以用 MPI_Bcast 函数，来取代 MPI_Send 和 MPI_Recv 函数。

回想在串行程序中，输入/输出参数是一个能够被调用函数使用和改变的值。然而对 MPI_Bcast 函数，data_p 参数在进程号为 source_proc 的进程中是一个输入参数，在其他进程中是一个输出参数。因此，当集合通信函数中的某个参数被标记为输入/输出（in/out）时，意味着可能在某些进程中它是输入参数，而在其他进程中是一个输出参数。

**程序 3-6　一个使用 MPI_Bcast 的 Get_input 函数版本**

```
1  void Get_input(
2        int      my_rank  /* in  */,
3        int      comm_sz  /* in  */,
4        double*  a_p      /* out */,
5        double*  b_p      /* out */,
6        int*     n_p      /* out */){
7
8     if (my_rank == 0){
9        printf("Enter a, b, and n\n");
10       scanf("%lf %lf %d", a_p, b_p, n_p);
11    }
12    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15 }  /* Get_input */
```

### 3.4.6　数据分发

如果我们想编写一个程序，用于计算向量和：

$$\begin{aligned}\boldsymbol{x}+\boldsymbol{y} &= (x_0,x_1,\cdots,x_{n-1})+(y_0,y_1,\cdots,y_{n-1})\\ &= (x_0+y_0,x_1+y_1,\cdots,x_{n-1}+y_{n-1})\\ &= (z_0,z_1,\cdots,z_{n-1})\\ &= \boldsymbol{z}\end{aligned}$$

如果用 double 类型的数组来表示向量，可以用串行加法来计算向量求和，代码如程序 3-7 所示。

**程序 3-7　向量求和的串行实现**

```
1  void Vector_sum(double x[], double y[], double z[], int n) {
2     int i;
3
4     for (i = 0; i < n; i++)
5        z[i] = x[i] + y[i];
6  }  /* Vector_sum */
```

如何用 MPI 实现这个程序呢？计算工作由向量的各个分量分别求和组成，所以我们可能只是指定各个任务求和的对应分量。这样，各个任务间没有通信，向量的并行加法问题就归结为聚合任务以及将它们分配到核上。如果分量的个数为 $n$，并且我们有 comm_sz 个核或者进程，那么可以简单地将连续 local_n 个向量分量所构成的块，分配到每个进程中。表 3-4 的左 4 列显示了当 $n=12$ 且 comm_sz = 3 时的例子。这种做法通常称为向量的**块划分**。

向量块划分的另一个方法是**循环划分**。在循环划分中，我们用轮转的方式去分配向量分量。表 3-4 的中间 4 列显示了当 $n=12$ 且 comm_sz = 3 时的例子。0 号进程得到了向量的 0 号分量，1 号进程得到了 1 号分量，2 号进程得到了 2 号分量，0 号进程又得到了 3 号分量，以此类推。

107 ~ 109

**表 3-4　在 3 个进程中，对有 12 个分量的向量的不同划分方式**

| 进程 | 分量 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 块划分 | | | | 循环划分 | | | | 块－循环划分块大小 =2 | | | |
| 0 | 0 | 1 | 2 | 3 | 0 | 3 | 6 | 9 | 0 | 1 | 6 | 7 |
| 1 | 4 | 5 | 6 | 7 | 1 | 4 | 7 | 10 | 2 | 3 | 8 | 9 |
| 2 | 8 | 9 | 10 | 11 | 2 | 5 | 8 | 11 | 4 | 5 | 10 | 11 |

第三种划分方法叫**块－循环划分**。基本思想是，用一个循环来分发向量分量所构成的块，而不是分发单个向量分量。所以首先要决定块的大小。如果 comm_sz = 3，$n$ = 12，并且块大小 $b$ = 2，块－循环划分就如表 3-4 右边的 4 列所显示的那样。

一旦决定如何划分向量，就能很容易地编写向量的并行加法函数：每个进程只要简单地将它所分配到的向量分量加起来。而且，无论使用哪种划分方法，每个进程都将有 local_n 个向量分量，为了节省存储空间，在每个进程上用一个只有 local_n 个元素的数组存储这些分量。因此，每个进程将运行程序 3-8 所显示的函数。虽然为了强调函数只对进程所分配到的部分向量进行操作，函数的名字有所改变，但这个函数本质上与原先串行函数的功能是一样的。

**程序 3-8　向量求和的并行实现**

```
1  void Parallel_vector_sum(
2        double  local_x[]  /* in  */,
3        double  local_y[]  /* in  */,
4        double  local_z[]  /* out */,
5        int     local_n    /* in  */) {
6     int local_i;
7
8     for (local_i = 0; local_i < local_n; local_i++)
9        local_z[local_i] = local_x[local_i] + local_y[local_i];
10 }  /* Parallel_vector_sum */
```

### 3.4.7　散射

现在假设我们想测试向量加法函数。先读取向量的维度，然后读取向量 x 和向量 y 是很方便 110
的。我们已知如何读取向量的维度：0 号进程提示用户，读取输入值，然后将值广播给其他进程。我们也可以用类似的方法读取向量：0 号进程读入向量，然后将它们广播给其他进程。但这种方法很浪费。如果有 10 个进程，向量有 1 万个分量，那么每个进程都需要为 1 万个分量的向量分配存储空间，但每个进程只在含有 1000 个分量的子向量上进行操作。例如，假设我们使用块划分法，那么如果 0 号进程只是将 1000 ~ 1999 号分量发送给 1 号进程，将 2000 ~ 2999 号分量分配给 2 号进程，以此类推。用这种方法，1 ~ 9 号进程将只需要为它们实际使用的向量分量分配存储空间。

因此，可以试着编写这样一个函数，0 号进程读入整个向量，但只将分量发送给需要分量的其他进程。MPI 提供了这样一个函数：

```
int MPI_Scatter(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      int           src_proc    /* in  */,
      MPI_Comm      comm        /* in  */);
```

如果通信子 comm 包含 comm_sz 个进程，那么 MPI_Scatter 函数会将 send_buf_p 所引用的数据分成 comm_sz 份，第一份给 0 号进程，第二份给 1 号进程，第三份给 2 号进程，以此类推。例如，假如我们使用块划分法，并且 0 号进程已经将一个有 $n$ 个分量的向量整个读入 send_buf_p 中，则 0 号进程将得到第一组 local_n = $n$/comm_sz 个分量，1 号进程将得到下一组 local_n 个分量，以此类推。每个进程应该将它本地的向量作为 recv_buf_p 参数的值，将 local_n 作为 recv_count 参数的值。send_type 和 recv_type 参数的值都应该是 MPI_DOUBLE，src_proc 参数的值应该是 0。令人惊讶的是：send_count 参数的值也应该是 local_n，因为 send

_count 参数表示的是发送到每个进程的数据量，而不是 send_buf_p 所引用的内存的数据量。如果使用块划分法和 MPI_Scatter 函数，我们可以如程序 3-9 所示，用 Read_vector 函数来读入向量。

需要注意的是，MPI_Scatter 函数将 send_count 个对象所组成的第一个块发送给了 0 号进程，将下一个由 send_count 个对象所组成的块发送给了 1 号进程，以此类推。所以，这种读取和分发输入向量的方法将只适用于块划分法，并且向量的分量个数 $n$ 可以整除 comm_sz 的情况。我们将在习题 18 中讨论处理循环划分以及块 - 循环划分法的部分解决方案，而完整的解决
111 方案参见 [23]。我们还将在习题 3.13 中讨论如何处理 $n$ 不能够整除 comm_sz 的情况。

**程序 3-9　一个读取并分发向量的函数**

```
1  void Read_vector(
2        double    local_a[]   /* out */,
3        int       local_n     /* in  */,
4        int       n           /* in  */,
5        char      vec_name[]  /* in  */,
6        int       my_rank     /* in  */,
7        MPI_Comm  comm        /* in  */) {
8
9     double* a = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       a = malloc(n*sizeof(double));
14       printf("Enter the vector %s\n", vec_name);
15       for (i = 0; i < n; i++)
16          scanf("%lf", &a[i]);
17       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18             MPI_DOUBLE, 0, comm);
19       free(a);
20    } else {
21       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22             MPI_DOUBLE, 0, comm);
23    }
24 }  /* Read_vector */
```

### 3.4.8　聚集

除非看见向量加法的结果，否则测试程序是无用的。所以我们需要编写一个可以打印分布式向量的函数。这个函数将向量的所有分量都收集到 0 号进程上，然后由 0 号进程将所有分量都打印出来。这个函数中的通信由 MPI_Gather 来执行，

```
int MPI_Gather(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      int           dest_proc   /* in  */,
      MPI_Comm      comm        /* in  */);
```

在 0 号进程中，由 send_buf_p 所引用的内存区的数据存储在 recv_buf_p 的第一个块中，在 1 号进程中，由 send_buf_p 所引用的内存区的数据存储在 recv_buf_p 的第二个块里，以此类推。所以，如果使用块划分法，就可以如同程序 3-10 所示，实现分布式向量打印函数。注意，
112 recv_count 指的是每个进程接收到的数据量，而不是所有接收到的数据量的总和。

使用 MPI_Gather 函数的限制与使用 MPI_Scatter 函数的限制是类似的：只有在使用块划分法，并且每个块的大小相同的情况下，打印函数才能正确运行。

**程序 3-10　一个打印分布式向量的函数**

```
1  void Print_vector(
2        double    local_b[]  /* in */,
3        int       local_n    /* in */,
4        int       n          /* in */,
5        char      title[]    /* in */,
6        int       my_rank    /* in */,
7        MPI_Comm  comm       /* in */) {
8
9     double* b = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       b = malloc(n*sizeof(double));
14       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
15             MPI_DOUBLE, 0, comm);
16       printf("%s\n", title);
17       for (i = 0; i < n; i++)
18          printf("%f ", b[i]);
19       printf("\n");
20       free(b);
21    } else {
22       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
23             MPI_DOUBLE, 0, comm);
24    }
25 }  /* Print_vector */
```

### 3.4.9　全局聚集

最后一个例子，我们来看看如何编写一个 MPI 程序，完成矩阵和向量的相乘。如果 $\boldsymbol{A} = (a_{ij})$ 是一个 $m \times n$ 的矩阵，$\boldsymbol{x}$ 是一个具有 $n$ 个分量的向量，那么 $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ 就是一个有 $m$ 个分量的向量。我们可以用 $\boldsymbol{A}$ 的第 $i$ 行与 $\boldsymbol{x}$ 的点积来求取 $\boldsymbol{y}$ 的第 $i$ 个分量：

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,n-1}x_{n-1}$$

见图 3-11。

因此，可以为串行矩阵乘法编写如下的伪代码。

```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

113

$$\begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & & a_{m-1,n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{pmatrix}$$

图 3-11　矩阵 – 向量乘法

事实上，这可以用 C 语言来编写。但 C 程序在处理二维数组时有一些特别之处（见习题 3.14），所以 C 语言的程序员常常用一维数组来“模拟”二维数组。最常见的做法是将一行内容存储在另一行后面。例如，二维数组，

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

作为一维数组会这样存储：

0 1 2 3 4 5 6 7 8 9 10 11

这个例子中，如果我们从0开始对行和列进行计数，那么存储在二维数组中的第2行第1列的元素（即9），它在一维数组中的位置为 $2 \times 4 + 1 = 9$。更为一般的情况是，如果数组有 $n$ 列，则当使用这种数组结构时，存储在第 $i$ 行第 $j$ 列的元素在一维数组中的位置就为 $i \times n + j$。使用这种一维数组结构，可以得到程序3-11中所示的C语言函数。

**程序3-11 矩阵－向量的串行乘法**

```
1  void Mat_vect_mult(
2        double  A[]  /* in  */,
3        double  x[]  /* in  */,
4        double  y[]  /* out */,
5        int     m    /* in  */,
6        int     n    /* in  */) {
7     int i, j;
8
9     for (i = 0; i < m; i++) {
10       y[i] = 0.0;
11       for (j = 0; j < n; j++)
12          y[i] += A[i*n+j]*x[j];
13    }
14 }  /* Mat_vect_mult */
```

如何并行化该程序？一个单独的任务可以是 $\boldsymbol{A}$ 的一个元素与 $\boldsymbol{x}$ 的一个分量相乘，并且将这个乘积加到 $\boldsymbol{y}$ 的一个分量上去。也就是说，下列语句的每一次执行是一个任务：

```
y[i] += A[i*n+j]*x[j];
```

所以，如果将y[i]分配给 $q$ 号进程，将A的第 $i$ 行也分配给 $q$ 号进程会很方便，也就是说，对A进行行划分。我们可以用块划分法、循环划分法或块－循环划分法来对行进行划分。在MPI中，使用块划分法是最简单的，所以这里对A的行进行块划分，并且与通常一样，假设comm_sz可以整除行数 $m$。

对A进行行划分可以使y[i]的计算包含所需要的A中的元素，所以对y也应该采用块划分。
114 即，如果A的第 $i$ 行分配给了 $q$ 号进程，$y$ 的第 $i$ 个分量也应该分配给 $q$ 号进程。

现在，y[i]的计算包含了A的第 $i$ 行中所有的元素，以及 $x$ 的所有分量，我们可以简单地将所有 $x$ 的分量分发给每个进程，来使通信量最小化。然而，在实际应用中，特别是矩阵为方阵时，使用矩阵－向量乘法函数的程序通常要执行多次乘法操作，并且从一次乘法操作得到的向量 $\boldsymbol{y}$ 的输出结果通常会是下一次迭代中向量 $\boldsymbol{x}$ 的输入。因此，实际上，通常假设对x的划分与对y的划分方法是相同的。

所以，如果x有一个块划分，我们该如何安排使得在执行下列循环前就使每个进程都能访问x中的所有分量呢？

```
for (j = 0; j < n; j++)
   y[i] += A[i*n+j]*x[j];
```

使用已经熟悉的集合通信，我们可以在执行一次MPI_Gather调用后，执行一次MPI_Bcast调用。在所有可能的情况下，这会涉及两个树形结构的通信，用蝶形通信结构可能会取得更好的效果。所以，MPI提供了这样的一个单独的函数：

```
int MPI_Allgather(
      void*          send_buf_p  /* in */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      MPI_Comm       comm        /* in  */);
```

这个函数将每个进程的 send_buf_p 内容串联起来，存储到每个进程的 recv_buf_p 参数中。通常，recv_count 指每个进程接收的数据量。所以大部分情况下，recv_count 的值与 send_ count 的值相同。

115

现在，我们就能够像程序 3-12 所示的那样实现矩阵 - 向量并行乘法。如果这个函数被多次调用，可以将 x 作为一个附加的参数传递给调用函数，这样能进一步提高性能。

**程序 3-12　MPI 矩阵 - 向量乘法函数**

```
1  void Mat_vect_mult(
2        double    local_A[]  /* in  */,
3        double    local_x[]  /* in  */,
4        double    local_y[]  /* out */,
5        int       local_m    /* in  */,
6        int       n          /* in  */,
7        int       local_n    /* in  */,
8        MPI_Comm  comm       /* in  */) {
9     double* x;
10    int local_i, j;
11    int local_ok = 1;
12
13    x = malloc(n*sizeof(double));
14    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15          x, local_n, MPI_DOUBLE, comm);
16
17    for (local_i = 0; local_i < local_m; local_i++) {
18       local_y[local_i] = 0.0;
19       for (j = 0; j < n; j++)
20          local_y[local_i] += local_A[local_i*n+j]*x[j];
21    }
22    free(x);
23 }  /* Mat_vect_mult */
```

## 3.5　MPI 的派生数据类型

在几乎所有的分布式内存系统中，通信比本地计算开销大很多。例如，从一个节点发送一个 double 类型的数据到另一个节点，耗费的时间比存储在节点本地内存里的两个 double 类型数据相加所耗费的时间长很多。而且，用多条消息发送一定数量的数据，明显比只用一条消息发送等量数据耗时。例如，我们可以预料到，下面的这对 for 循环比单个的发送/接收对要慢得多：

116

```
double x[1000];
. . .
if (my_rank == 0)
   for (i = 0; i < 1000; i++)
      MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
   for (i = 0; i < 1000; i++)
      MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);

if (my_rank == 0)
   MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else  /* my_rank == 1 */
   MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

实际上，在某个系统里，发送和接收循环花费将近 50 倍的时间。在另一个系统里，循环代码的执行花费超过 100 倍的时间。因此，如果减少发送的消息数量，就能够提高程序的性能。

MPI 提供了三个基本手段来整合可能需要多条消息的数据：不同通信函数中的 count 参数、派生数据类型，以及 MPI_Pack/Unpack 函数。我们已经介绍过 count 参数，它可以用于将连续的数组元素集合起来组成一条单独的消息。本节我们将讨论创建派生数据类型的方法。在习题中，我们将看到创建派生数据类型的其他方法以及使用 MPI_Pack/Unpack 函数的方法。

在 MPI 中，通过同时存储数据项的类型以及它们在内存中的相对位置，**派生数据类型**可以用于表示内存中数据项的任意集合。其主要思想是：如果发送数据的函数知道数据项的类型以及在内存中数据项集合的相对位置，就可以在数据项被发送出去之前在内存中将数据项聚集起来。类似地，接收数据的函数可以在数据项被接收后将数据项分发到它们在内存中正确的目标地址上。例如，在梯形积分法程序中，需要调用 MPI_Bcast 函数三次：一次广播左端点 $a$、一次广播右端点 $b$、一次广播梯形的个数 $n$。另一种替代方法是：建立一个单独的派生数据类型，该数据类型由两个 double 类型数据和一个 int 类型数据组成。这样，只需要调用 MPI_Bcast 一次。在 0 号进程中，$a$、$b$ 和 $n$ 在一次函数调用中被发送出去；而在其他进程中，这些值将在函数调用中被接收。

正式地，一个派生数据类型是由一系列的 MPI 基本数据类型和每个数据类型的偏移所组成的。在梯形积分法的例子中，假设在 0 号进程里变量 $a$、$b$ 和 $n$ 在内存中的位置为如下的地址：

| 变量 | 地址 |
|---|---|
| a | 24 |
| b | 40 |
| n | 48 |

那么下面的派生数据类型就可以表示这些数据项：

117 {(MPI_DOJBLE,0),(MPI_DOUBLE,16),(MPI_INT,24)}.

每一对数据项的第一个元素表明数据类型，第二个元素是该数据项相对于起始位置的偏移。假设派生类型从 a 开始，则 a 的偏移为 0，其他元素的偏移从 a 的起始位置开始算，偏移量以字节为单位。数据项 b 距离 a 的偏移是 40 − 24 = 16 字节，数据项 c 距离 a 的偏移是 n = 48 − 24 = 24 字节。

我们可以用 MPI_Type_create_struct 函数创建由不同基本数据类型的元素所组成的派生数据类型：

```
int MPI_Type_create_struct(
      int             count                          /* in  */,
      int             array_of_blocklengths[]        /* in  */,
      MPI_Aint        array_of_displacements[]       /* in  */,
      MPI_Datatype    array_of_types[]               /* in  */,
      MPI_Datatype*   new_type_p                     /* out */);
```

参数 count 指的是数据类型中元素的个数，所以在我们的这个例子中，它应该为 3。每个数组参数都有 count 个元素。第一个数组，array_of_blocklengths 允许单独的数据项可能是一个数组或者子数组。例如，如果第一个元素是一个含 5 个元素的数组，那么有：

```
array_of_blocklengths[0] = 5;
```

但在我们的例子中，没有元素是数组，所以可以简单地定义：

```
int array_of_blocklengths[3] = {1, 1, 1};
```

MPI_Type_create_struct 的第三个参数 array_of_displacements 指定了距离消息起始位置的偏移量，单位为字节。所以有：

```
array_of_displacements[] = {0, 16, 24};
```

为了找到这些值，可以使用MPI_Get_address函数：

```
int MPI_Get_address(
      void*       location_p  /* in  */,
      MPI_Aint*   address_p   /* out */);
```

它返回的是location_p所指向的内存单元的地址。这个特殊类型的MPI_Aint是整数型，它的长度足以表示系统地址。因此，为了取得array_of_displacements里的各个值，我们可以用下面的代码：

```
MPI_Aint a_addr, b_addr, n_addr;

MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0;
MPI_Get_address(&b, &b_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Get_address(&n, &n_addr);
array_of_displacements[2] = n_addr - a_addr;
```

118

array_of_datatypes存储的是元素的MPI数据类型，我们可以定义：

```
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
```

在这些初始化工作完成之后，就可以通过函数调用建立新的数据类型：

```
MPI_Datatype input_mpi_t;
. . .
MPI_Type_create_struct(3, array_of_blocklengths,
      array_of_displacements, array_of_types,
      &input_mpi_t);
```

在使用通信函数中的input_mpi_t之前，我们必须先用一个函数调用去指定它：

```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

它允许MPI实现为了在通信函数内使用这一数据类型，优化数据类型的内部表示。

现在，为了使用new_ mpi_ t这个新的数据类型，需要在每个进程上调用MPI_Bcast：

```
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

所以，我们可以像使用MPI的基本数据类型一样去使用input_mpi_t类型。

在构造新数据类型的过程中，MPI实现可能要在内部分配额外的存储空间。因此，当我们使用新的数据类型时，可以用一个函数调用去释放额外的存储空间：

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

这里，我们采用上述步骤定义了可以被Get_input函数调用的Build_mpi_type函数。这一新的函数以及修改过的Get_input函数见程序3-13。

## 3.6 MPI程序的性能评估

接下来，我们分析矩阵-向量乘法程序的性能。在编写程序时，尽可能使程序并行化，因为我们希望解决相同问题时，并行程序比串行程序运行得更快。那么该如何证明这一点呢？我们在2.6节中已经讨论了这个问题，所以我们先回顾一些在那里学过的知识。

### 3.6.1 计时

通常，我们不会对程序从开始运行到结束运行所耗费的时间感兴趣。例如，在矩阵-向量乘法中，我们一般不会对输入矩阵和输出乘积结果所花费的时间感兴趣，而只对实际的乘法运算所花费的时间感兴趣。所以需要修改源代码，加入函数调用，统计从乘法运算开始到结束所经过的

119

时间。MPI 提供了这样的一个函数，MPI_Wtime，它返回从过去某一时刻开始所经过的秒数：

```
double MPI_Wtime(void);
```

**程序 3-13　使用派生数据类型的 Get_input 函数**

```
void Build_mpi_type(
      double*         a_p             /* in  */,
      double*         b_p             /* in  */,
      int*            n_p             /* in  */,
      MPI_Datatype*   input_mpi_t_p   /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};

   MPI_Get_address(a_p, &a_addr);
   MPI_Get_address(b_p, &b_addr);
   MPI_Get_address(n_p, &n_addr);
   array_of_displacements[1] = b_addr-a_addr;
   array_of_displacements[2] = n_addr-a_addr;
   MPI_Type_create_struct(3, array_of_blocklengths,
         array_of_displacements, array_of_types,
         input_mpi_t_p);
   MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */

void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

对一个 MPI 代码块进行计时：

119
~
120

```
double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
      my_rank, finish-start);
```

计算串行代码运行的时间不需要连接 MPI 库。在 POSIX 库中也有一个名为 gettimeofday 的函数，它返回自过去的某一时间点到计时点经历了多少毫秒。具体的语法并不重要。在头文件 timer.h 中提供了一个宏 GET_TIME，你可以到本书的网站下载该文件。此宏需要传入的参数是 double 类型的：

```
#include "timer.h"
. . .
double now;
. . .
GET_TIME(now);
```

执行 GET_TIME 后，now 参数中会存储从过去到现在经过的时间。如果你想要该宏计算串行代码

所经历的毫秒级时间，可以运行下面的代码：

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```

有一点需要强调的是：GET_TIME是一个宏，所以代码在预编译时被直接插入源代码中。因此，它是直接对double型的参数进行操作，而不是对指向double类型数据的指针进行操作。此外，timer.h不在系统库的默认路径内，如果它也不在你编译程序的当前文件夹内，那么此时就有必要告诉编译器去哪儿找到它。例如，timer.h在/home/peter/my_include路径下，应使用以下命令来编译一个使用GET_TIME函数的程序：

```
$ gcc -g -Wall -I/home/peter/my_include -o <executable>
      <source_code.c>
```

MPI_Wtime和GET_TIME都返回墙上时钟时间。回想一下，C语言中的clock函数返回的是CPU时间（包括用户代码、库函数以及系统调用函数所消耗的时间），但它不包括空闲时间，而在并行程序中，很多情况下都是空闲等待状态。例如，调用MPI_Recv会消耗很多时间来等待消息的到达。而墙上时钟时间给出了所经历的全部时间，包括空闲等待时间。 [121]

还剩下一些问题有待解决。首先，如前所述，并行程序会为每个进程报告一次comm_sz时间，但我们需要获得一个总的单独时间。理想情况是，所有的进程同时开始运行矩阵乘法，当最后一个进程完成运算时，能获取从开始到最后一个进程结束之间的时间开销。换句话讲，并行时间取决于“最慢”进程花费的时间。这一时间并不是完全精确的，因为我们无法保证所有进程都开始于同一个时间点，但也足够接近理想的衡量时间。MPI的集合通信函数MPI_Barrier能够确保同一个通信子中的所有进程都完成调用该函数之前，没有进程能够提前返回。它的语法是：

```
int MPI_Barrier(MPI_Comm   comm   /* in */);
```

下面这段代码可以用来对一段MPI程序进行计时并报告运行时间：

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

其中MPI_Reduce函数使用MPI_MAX运算符，在所有输入参数中找出最大的local_elapsed。

第2章提到，我们需要注意每次计时的变化。尽管每次的输入参数一样，进程数相同，运行环境也没有发生改变，但多次运行同一段程序，仍然可能会见到运行时间有变化。因为系统其余部分，尤其是操作系统的影响，是不可预知的。这一影响的存在，使得程序不可能运行得比在“无干扰”系统中更快，我们通常会报告最小运行时间而不是平均时间（想得到更多的信息，请看[5]）。

最后，当 MPI 程序运行在多核处理器的混合系统上时，在每个节点上只运行一个 MPI 进程。这可以减少交互时的竞争，从而获得更佳运行时间，也能减少运行时的变化。

### 3.6.2 结果

122 表 3-5 是矩阵 - 向量乘法程序的计时结果。输入矩阵是一个方阵。时间以毫秒为单位，并保留 2 位有效位。comm_sz 为 1 的时间表示在分布式内存系统中单个核上运行串行程序的时间。显然，如果固定 comm_sz 的值，增大 $n$，那么矩阵的大小和程序的运行时间也会增加。进程数相对较少时，$n$ 增大 1 倍就会使运行时间变为原来的 4 倍，然而当进程数很多时，此公式就不成立了。

**表 3-5 矩阵 - 向量乘法的串行和并行程序的运行时间（单位：毫秒）**

| | 矩阵的秩 | | | | |
|---|---|---|---|---|---|
| comm_sz | 1024 | 2048 | 4096 | 8192 | 16 384 |
| 1 | 4.1 | 16.0 | 64.0 | 270 | 1100 |
| 2 | 2.3 | 8.5 | 33.0 | 140 | 560 |
| 4 | 2.0 | 5.1 | 18.0 | 70 | 280 |
| 8 | 1.7 | 3.3 | 9.8 | 36 | 140 |
| 16 | 1.7 | 2.6 | 5.9 | 19 | 71 |

如果我们固定 $n$、增大 comm_sz，那么运行时间会减少。事实上，对于值很大的 $n$ 来说，进程数加倍大约能减少一半的运行时间。然而，对值小的 $n$，增大 comm_sz 获得的效果甚微，例如，当 $n=1024$ 时，进程数从 8 增大到 16 后，运行时间没有出现变化。

这些都是非常典型的并行运行时间，当我们增大问题的规模，运行时间也随之变大，无论进程数多少，都是这个趋势。增加的速率可以较为匀速（比如只运行一个进程的时候），也可能发生剧烈变化（比如运行 16 个进程的情况）。当增加进程数时，运行时间会在一个阶段内减小。然而，达到某点后运行时间可能会开始变得很慢。在 1024 阶矩阵的情况下，使进程数从 8 增到 16 时，我们就遇到这个情况。

这种现象的原因是：串行程序的运行时间与对应的并行程序的运行时间有共同的联系。回忆一下，$T_{串行}$代表串行时间，由于它取决于输入值 $n$，所以把它设为 $T_{串行}(n)$。同理，并行运行时间 $T_{并行}$取决于输入值 $n$ 和进程数目 comm_sz $=p$，则设为 $T_{并行}(n, p)$。第 2 章曾经提到，并行程序会将串行程序的工作分配到各个进程上，但又会增加额外的开销，设此开销为：

$$T_{并行}(n,p) = T_{串行}(n)/p + T_{开销}$$

123 在 *MPI* 程序中，并行计算的开销一般来自于通信，它同时还受到问题集的规模和进程数的影响。

我们的矩阵 - 向量乘法程序并不难实现，主要的串行运算部分是一对 for 循环的嵌套：

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i*n+j]*x[j];
}
```

如果只对浮点运算进行计数，则内层循环会执行 $n$ 次乘法和 $n$ 次加法，总共 $2n$ 次浮点运算。因为执行了 $m$ 次内层循环，则执行上面这段代码总共需要 $2mn$ 次运算。所以当 $m=n$ 时：

$$T_{串行}(n) \approx an^2$$

$a$ 是常数（符号≈意味着近似等于）。

如果串行程序要执行 $n \times n$ 矩阵与一个 $n$ 维向量相乘，那么并行程序中每个进程进行 $n/p \times n$

矩阵与 $n$ 维向量相乘。每一个局部的矩阵－向量相乘要执行 $n^2/p$ 次浮点运算，即每个进程的工作量被削减了 $p$ 倍。

然而，并行程序在进行本地矩阵－向量乘法前，需要调用 MPI_Allgather 函数。在我们的例子中：

$$T_{并行}(n,p) = T_{串行}(n)/p + T_{allgather}$$

而且，根据对计时数据的观察发现，$p$ 值较小且 $n$ 值较大时，公式中起主导地位的是 $T_{串行}(n)/p$。假定初始时 $p$ 较小（如 $p=2$、4），然后将 $p$ 增加1倍，大约可以令运行时间减少一半，例如：

$$T_{串行}(4096) = 1.9 \times T_{并行}(4096,2)$$
$$T_{串行}(8192) = 1.9 \times T_{并行}(8192,2)$$
$$T_{并行}(8192,2) = 2.0 \times T_{并行}(8192,4)$$
$$T_{串行}(16,384) = 2.0 \times T_{并行}(16,384,2)$$
$$T_{并行}(16,384,2) = 2.0 \times T_{并行}(16,384,4)$$

而且，如果将 $p$ 固定在较小值（如 $p=2$、4），然后增加 $n$，效果似乎与增加串行程序中 $n$ 的效果差不多。例如：

$$T_{串行}(4096) = 4.0 \times T_{串行}(2048)$$
$$T_{并行}(4096,2) = 3.9 \times T_{并行}(2048,2)$$
$$T_{并行}(4096,4) = 3.5 \times T_{并行}(2048,4)$$
$$T_{串行}(8192) = 4.2 \times T_{串行}(4096)$$
$$T_{并行}(8192,2) = 4.2 \times T_{并行}(4096,2)$$
$$T_{并行}(8192,4) = 3.9 \times T_{并行}(8192,4)$$

124

这些观查结果表明：并行运算时间与串行运算时间差不多（即 $T_{并行}(n,p)$ 约等于 $T_{串行}(n)/p$）。所以 $T_{allgather}$ 对性能没有造成多大影响。

另一方面，当 $n$ 值较小、$p$ 值较大时，上面得出的结论不成立，例如：

$$T_{并行}(1024,8) = 1.0 \times T_{并行}(1024,16)$$
$$T_{并行}(2048,16) = 1.5 \times T_{并行}(1024,16)$$

可以看出：当 $n$ 值较小、$p$ 值较大时，公式中的对 $T_{并行}$ 起主导因素的参数是 $T_{allgather}$。

### 3.6.3　加速比和效率

**加速比**经常用来衡量串行运算和并行运算时间之间的关系，它表示为串行时间与并行时间的比值：

$$S(n,p) = \frac{T_{串行(n)}}{T_{并行(n,p)}}$$

$S(n, p)$ 最理想的结果是 $p$。如果 $S(n,p)=p$，说明拥有 comm_sz $=p$ 个进程数的并行程序能运行得比串行程序快 $p$ 倍。这种被我们称为**线性加速比**的情况事实上很少出现。表3-6给出了矩阵－向量乘法程序各种情况下的加速比结果。在 $p$ 较小、$n$ 较大的情况下，我们获得了近似于线性的加速比，然而，当 $p$ 较大、$n$ 较小时，加速比则远远小于 $p$。最差的一种情况是 $n=1024$ 和 $p=16$ 时，只得到了2.4的加速比。

另外，并行的**效率**也是评价并行性能的重要指标之一，它其实是“每个进程”的加速比：

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{串行(n)}}{p \times T_{并行(n,p)}}$$

线性加速比相当于并行效率 $p/p=1.0$，通常，效率都小于1。

表 3-6 并行矩阵 – 向量乘法的加速比

| comm_sz | 矩阵的秩 | | | | |
|---|---|---|---|---|---|
| | 1024 | 2048 | 4096 | 8192 | 16 384 |
| 1 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 2 | 1.8 | 1.9 | 1.9 | 1.9 | 2.0 |
| 4 | 2.1 | 3.1 | 3.6 | 3.9 | 3.9 |
| 8 | 2.4 | 4.8 | 6.5 | 7.5 | 7.9 |
| 16 | 2.4 | 6.2 | 10.8 | 14.2 | 15.5 |

125

表 3-7 并行矩阵 – 向量乘法的效率

| comm_sz | 矩阵的秩 | | | | |
|---|---|---|---|---|---|
| | 1024 | 2048 | 4096 | 8192 | 16 384 |
| 1 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 2 | 0.89 | 0.94 | 0.97 | 0.96 | 0.98 |
| 4 | 0.51 | 0.78 | 0.89 | 0.96 | 0.98 |
| 8 | 0.30 | 0.61 | 0.82 | 0.94 | 0.98 |
| 16 | 0.15 | 0.39 | 0.68 | 0.89 | 0.97 |

表 3-7 罗列了矩阵 – 向量乘法程序的并行效率。再一次强调，在 $p$ 较小、$n$ 较大的情况下，有近似线性的效率；相反，在 $p$ 较大、$n$ 较小的情况下，远远达不到线性效率。

### 3.6.4 可扩展性

我们的矩阵 – 向量乘法的程序无法在 $n$ 较小、$p$ 较大时取得线性加速比，那是不是意味着这不是一个好程序呢？许多计算机科学家用“可扩展性”来回答这问题。粗略地讲，如果问题的规模以一定的速率增大，但效率没有随着进程数的增加而降低，那么就可以认为程序是可扩展的。

这个定义的争议之处在于“问题规模以一定的速率增大……”试想两个并行程序 A 和 B，假设 $p \geqslant 2$，不考虑问题的规模，程序 A 的效率是 0.75。另外一个并行程序 B，$p \geqslant 2$ 且 $1000 \leqslant n \leqslant 625p$，程序 B 的效率为 $n/(625p)$。根据我们的“定义”，两个程序都是可扩展的。对于程序 A，维持效率所需要的问题规模递增速率为 0；而对于程序 B 来说，如果增加 $n$ 的速率与增加 $p$ 一样快，那么就能够维持一个恒定的效率。例如，$n = 1000$ 和 $p = 2$ 时，B 的效率为 0.80，如果把 $p$ 的数值翻倍为 4，同时问题的规模仍然维持 $n = 1000$，那么程序的效率将会降到 0.40，但是，若我们同时把 $n$ 也增加 1 倍到 2000，那么整个程序的效率就能继续维持在 0.80。所以从这点来看，程
126 序 A 比程序 B 更有扩展性，但是两者都满足我们对于可扩展的定义。

再看表 3-7 中的并行效率，可以看到矩阵 – 向量乘法程序没能达到与程序 A 一样高的可扩展性：在大多数情况下，当 $p$ 增加时，效率就会降低。另一方面，这个并行程序有些类似于程序 B：当 $p \geqslant 2$，以 2 为倍数同时递增 $p$ 和 $n$，并行效率确实提高了，唯一的例外发生在 $p$ 从 2 增到 4 时。计算机科学家在谈及可扩展性时，一般主要关注 $p$ 数值非常大的情况下。当 $p$ 从 4 增大到 8，或者从 8 增大到 16，$n$ 以 2 的倍数增加时，程序的并行效率也会增加。

若程序可以在不增加问题规模的前提下维持恒定效率，那么此程序称为拥有**强可扩展性**；当问题规模增加，通过增大进程数来维持程序效率的，称为**弱可扩展性**。程序 A 是前者，程序 B 是后者，我们的矩阵 – 向量乘法显然也是弱可扩展性的。

## 3.7 并行排序算法

在分布式内存系统上如何实现并行排序算法？“输入”和“输出”分别是什么？答案取决于需要排序的键值存储在哪。可以在程序开始或结束时，将键值分布在多个进程中，也可以只分配给一个进程。本节，我们来讨论一个分布式算法的实现，其中键值是分配到各个进程上。在编程作业3.8中，我们将探讨结束时将键值分配给一个进程的算法。

设总共有 $n$ 个键值，p = comm_sz 个进程，给每个进程分配 $n/p$ 个键值（与前面一样，假定 $n$ 能被 $p$ 整除）。开始时，不对哪些键值分配到哪个进程上加以限制，然而在算法结束时：

- 每个进程上的键值应该以升序的方式存储。
- 若 $0 \leqslant q < r < p$，则分配给进程 $q$ 的每一个键值应该小于等于分配给进程 $r$ 的每一个键值。

所以，如果按照进程编号来进行键值排列（即先是进程0的键值，接着进程1的，以此类推），所有的键值就可以按升序排列。为了保证表达的清晰性，假设键值都是普通的 int 类型。

### 3.7.1 简单的串行排序算法

开始排序前，让我们先看看一组简单的串行排序算法，可能其中最有名的就是冒泡排序法（见程序3-14）。数组 a 存储未排序的键值，调用排序函数后给出排完序的键值。数组 a 中共有 n 个键值，算法按对比较元素大小：a[0]与a[1]比，a[1]和a[2]比，以此类推，只要该对的顺序不对，就互相交换位置。当 list_length = n 时，第一次外部循环遍历后，序列中的最大值被移动到 a[n-1]。第二回遍历去除最后一个元素，并把次大的元素移入 a[n-2]。所以，随着 list_length 的减少，越来越多排好序的元素安置在数组的后部。 [127]

**程序3-14 串行冒泡排序**

```
1  void Bubble_sort(
2          int  a[]  /* in/out */,
3          int  n    /* in     */) {
4      int list_length, i, temp;
5
6      for (list_length = n; list_length >= 2; list_length--)
7          for (i = 0; i < list_length-1; i++)
8              if (a[i] > a[i+1]) {
9                  temp = a[i];
10                 a[i] = a[i+1];
11                 a[i+1] = temp;
12             }
13
14 }  /* Bubble_sort */
```

因为其固有的串行按对排序特性，并行化上述算法意义不大。假设 a[i-1]=9，a[i]=5，a[i+1]=7，算法会先比较9和5并交换，然后比较9和7并交换，得到5、7、9的排列。但是，如果比较操作本身是乱序的，例如，先比较5和7，再比较9和5，比较后得到的序列是5、9、7。因此，“比较-交换”的顺序对算法的正确性非常重要。

冒泡排序的一个变种是**奇偶交换排序**，该算法更加适合并行化。关键在于去耦的比较-交换。此算法由一系列阶段组成，这些阶段分2种类型。在偶数阶段，比较-交换由以下数对执行：

$$(a[0],a[1]),(a[2],a[3]),(a[4],a[5]),\cdots,$$

而奇数阶段则由以下数对进行比较-交换：

$$(a[1],a[2]),(a[3],a[4]),(a[5],a[6]),\cdots,$$

这里有个小例子：

开始：5、9、4、3

偶数阶段：比较－交换（5，9）和（4，3），获得序列5、9、3、4。

奇数阶段：比较－交换（9，3），获得序列5、3、9、4。

偶数阶段：比较－交换（5，3）和（9，4），获得序列3、5、4、9。

[128] 奇数阶段：比较－交换（5，4），获得序列3、4、5、9。

这个例子需要四个阶段来排序四个元素的列表。一般来说，阶段可能会更少些，下面的这个定理保证了我们至多用 $n$ 个阶段排序 $n$ 个元素。

**定理**：设 $A$ 是一个拥有 $n$ 个键值的列表，作为奇偶交换排序算法的输入，那么经过 $n$ 个阶段后，$A$ 能够排好序。

程序3-15显示了一个串行奇偶交换排序函数。

**程序 3-15　奇偶排序程序的串行代码**

```
1  void Odd_even_sort(
2         int  a[]  /* in/out */,
3         int  n    /* in     */) {
4     int phase, i, temp;
5
6     for (phase = 0; phase < n; pnase++)
7        if (phase % 2 == 0) { /* Even phase */
8           for (i = 1; i < n; i += 2)
9              if (a[i-1] > a[i]) {
10                temp = a[i];
11                a[i] = a[i-1];
12                a[i-1] = temp;
13             }
14       } else { /* Odd phase */
15          for (i = 1; i < n-1; i += 2)
16             if (a[i] > a[i+1]) {
17                temp = a[i];
18                a[i] = a[i+1];
19                a[i+1] = temp;
20             }
21       }
22 }  /* Odd_even_sort */
```

### 3.7.2　并行奇偶交换排序

很清楚的是：奇偶交换排序远比冒泡排序适合并行化，因为在一个阶段内所有的比较－交换都能同时进行。

实现 Foster 方法有很多可能的办法，这里是其中一个：

- 任务：在阶段 $j$ 结束时确定 a[i]的值
- 通信：确定 a[i]值的任务需要与其他确定 a[i+1]或者 a[i-1]的任务进行通信，同时，在阶段 $j$ 结束时，a[i]的值需要用来在阶段 $j+1$ 结束时确定 a[i]的值。

[129] 图3-12显示了这一过程，我们用 a[i]来标记确定 a[i]值的任务。

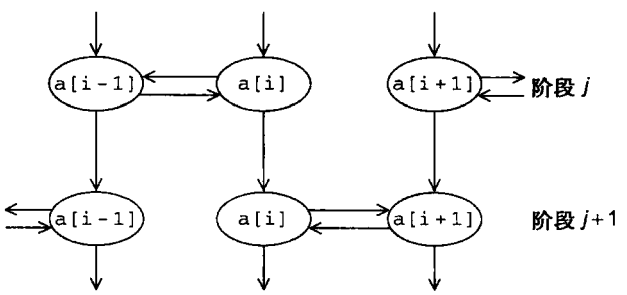



图3-12　一次奇偶排序中任务间的通信。用 a[i]来标记确定 a[i]值的任务

我们注意到：在排序算法开始和结束阶段，给每个进程分配 $n/p$ 个键值。因此，在这个情况下，算法中的聚集和分配部分是由问题的描述来指定的。让我们看看以下两种情况。

当 $n=p$ 时，图3-12非常清楚地描述了算法是如何进行的。根据阶段，进程 $i$ 可以将当前自己拥有 a[i]值发送给进程 $i-1$ 或者进程 $i+1$。同时，它也应该接收存储在进程 $i-1$ 或者进程 $i+1$ 的值，然后决定保留两个值中的哪个来为下一个阶段做准备。

然而，事实上，不太可能在 $n=p$ 的情况下运行该算法，因为不可能有成百上千个处理器。另外，即使有这样的处理器，每一次比较-交换时，发送和接收消息导致的额外时间开销会严重降低程序的运行时间。记住，通信开销远比“局部”计算（如“比较”）开销大。

当每个进程存储 $n/p>1$ 个元素时（$n$ 能被 $p$ 整除），该怎样修改程序呢？来看看下面的例子，假设我们有 $p=4$ 个进程，$n=16$ 个键值，详见表3-8。对分配给每个进程的键值使用快速的串行排序法，如用C语言库的 qsort 函数对局部链值进行排序。现在，如果每个进程有一个元素，那么进程0和进程1交换数据，进程2和进程3交换数据。我们试着使进程0和进程1交换数据，进程2和进程3交换数据，那么进程0就会有4个比进程1中数据小的元素，而进程2则拥有4个比进程3中数据小的元素，这就是表3-8中第3行的情况。再次看一个进程中只有一个元素的例子，在阶段1，进程1和进程2交换元素，而进程0和进程3是空闲的。如果进程1有着较小的一些元素，而进程2拥有较大的一些元素，那么就得到表3-8中第4行的情况。继续让这些进程再运行2个阶段，就能获得排好序的列表。即每个进程的键值以升序排列，并且，如果 $q<r$，分配给进程 $q$ 的键值必定小于或等于分配进程 $r$ 的键值。 130

事实上，我们所举的例子已经是该算法性能最差时的情况了。

**表3-8　并列奇偶交换排序**

| 时间 | 进程 | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| 开始 | 15, 11, 9, 16 | 3, 14, 8, 7 | 4, 6, 12, 10 | 5, 2, 13, 1 |
| 局部排序后 | 9, 11, 15, 16 | 3, 7, 8, 14 | 4, 6, 10, 12 | 1, 2, 5, 13 |
| 阶段0后 | 3, 7, 8, 9 | 11, 14, 15, 16 | 1, 2, 4, 5 | 6, 10, 12, 13 |
| 阶段1后 | 3, 7, 8, 9 | 1, 2, 4, 5 | 11, 14, 15, 16 | 6, 10, 12, 13 |
| 阶段2后 | 1, 2, 3, 4 | 5, 7, 8, 9 | 6, 10, 11, 12 | 13, 14, 15, 16 |
| 阶段3后 | 1, 2, 3, 4 | 5, 6, 7, 8 | 9, 10, 11, 12 | 13, 14, 15, 16 |

**定理**：*如果由 $p$ 个进程运行并行奇偶交换排序算法，则 $p$ 个阶段后，输入列表排序完毕。*

并行算法对于人工计算机来说已足够清晰：

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

然而，在把该算法转为MPI程序前，我们需要先搞清楚一些细节问题。

首先，怎么计算配对运行的另一个进程的进程号？当一个进程空闲时，它的配对进程又是什

么？在偶数阶段，进程号是奇数号进程与 my_rank - 1 号进程配对进行交换，而进程号是偶数的进程与 my_rank + 1 号进程配对进行交换；在奇数阶段时，上述配对正好相反。然而，这种配对可能是无效的：如果 my_rank = 0 或者 my_rank = comm_sz - 1，那么它的配对进程号就是 -1 或者 comm_sz。若 partner = -1 或 partner = comm_sz，那么该进程应该是空闲的。可以计算 Compute_partner 来决定一个进程是否空闲。

```
if (phase % 2 == 0)          /* Even phase */
   if (my_rank % 2 != 0)     /* Odd rank */
      partner = my_rank - 1;
   else                      /* Even rank */
      partner = my_rank + 1;
131 else                     /* Odd phase */
   if (my_rank % 2 != 0)     /* Odd rank */
      partner = my_rank + 1;
   else                      /* Even rank */
      partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
   partner = MPI_PROC_NULL;
```

MPI_PROC_NULL 是由 MPI 库定义的一个常量。在点对点通信中，将它作为源进程或者目标进程的进程号，此时，调用通信函数后会直接返回，不会产生任何通信。

### 3.7.3 MPI 程序的安全性

如果进程不是空闲的，我们可以通过调用 MPI_Send 和 MPI_Recv 来实现通信：

```
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,
      MPI_STATUS_IGNORE);
```

但这可能会导致程序挂起或者崩溃。回忆一下，MPI 标准允许 MPI_Send 以两种不同的方式来实现：简单地将消息复制到 MPI 设置的缓冲区并返回，或者直到对应的 MPI_Recv 出现前都阻塞。此外，许多 MPI 函数都设置了使系统从缓冲到阻塞间切换的阈值，即相对较小的消息就交由 MPI_Send 缓冲，但对于大型数据就选择阻塞模式。如果每个进程都阻塞在 MPI_Send 上，则没有进程会去调用 MPI_Recv，此时程序就会**死锁**或挂起，每个进程都在等待一个不会发生的事件发生。

依赖于 MPI 提供的缓冲机制是**不安全的**，这样的程序在运行一些输入集时没有问题，但有可能在运行其他输入集时导致崩溃或挂起。这样使用 MPI_Send 和 MPI_Recv，程序会不安全。可能当 $n$ 值较小时，程序没有问题，但一旦 $n$ 变大，问题或许就会出现从而导致挂起或崩溃现象。

这里引出了一些问题：

1）一般来说，怎么才能说一个程序是安全的？

2）我们怎样修改并行奇偶交换排序程序的通信过程，使其安全？

要解决第 1 个问题，我们可以用 MPI 标准提供的另一个函数来代替 MPI_Send，这个函数是 MPI_Ssend。这个额外的字母“s”代表同步，函数 MPI_Ssend 保证了直到对应的接收开始前，发送端一直阻塞。所以，我们通过将 MPI_Send 替换为 MPI_Ssend 来检查程序是否安全，如果输入合适的值和 comm_sz，程序没有挂起或者崩溃，那么原来的程序是安全的。MPI_Ssend 与
132 MPI_Send 调用的参数是相同的。

```
int MPI_Ssend(
      void*         msg_buf_p     /* in */,
      int           msg_size      /* in */,
      MPI_Datatype  msg_type      /* in */,
      int           dest          /* in */,
      int           tag           /* in */,
      MPI_Comm      communicator  /* in */);
```

第二个问题的解决方法是重构通信。造成程序不安全的最大因素在于多个进程先同时发送消息，再同时接收消息。我们在配对进程之间的数据交换就是其中一个例子。另一个例子是“环状传递”，每个进程 $q$ 向进程 $q+1$ 发送消息，进程 comm_sz − 1 向进程 0 发送消息：

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
      0, comm, MPI_STATUS_IGNORE);
```

我们需要重构这两个通信函数，使一些进程先接收消息再发送消息。例如，可以改成：

```
if (my_rank % 2 == 0) {
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE);
} else {
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE);
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}
```

如果 comm_sz 是偶数，那么这样的改动会取得很好效果。例如 comm_sz = 4，则进程 0 和进程 2 会先向进程 1 和进程 3 发送消息，而进程 1 和进程 3 等待接收来自进程 0 和进程 2 的消息。而在下一个发送 − 接收阶段，前后两对进程换一个次序接收和发送消息：进程 1 和进程 3 发送消息给进程 2 和进程 0，而进程 2 和进程 0 接收来自进程 1 和 3 的消息。

但是，当 comm_sz 是奇数时（comm_sz > 1），这个机制可能是不安全的。假定 comm_sz = 5。图 3-13 显示了事件另一种可能的顺序，实线箭头表明完整的通信，虚线箭头表示该通信正在等待完成。

MPI 提供了自己调度通信的方法，我们把这个函数称为 MPI_Sendrecv：

```
int MPI_Sendrecv(
      void*          send_buf_p      /* in  */,
      int            send_buf_size   /* in  */,
      MPI_Datatype   send_buf_type   /* in  */,
      int            dest            /* in  */,
      int            send_tag        /* in  */,
void*          recv_buf_p      /* out */,
int            recv_buf_size   /* in  */,
MPI_Datatype   recv_buf_type   /* in  */,
int            source          /* in  */,
int            recv_tag        /* in  */,
MPI_Comm       communicator    /* in  */,
MPI_Status*    status_p        /* in  */);
```

133

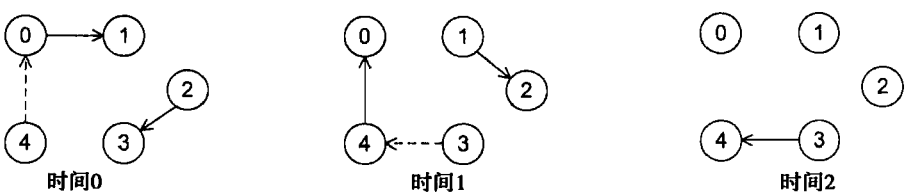



图 3-13　5 个进程之间的安全通信

调用一次这个函数，它会分别执行一次阻塞式消息发送和一次消息接收，dest 和 source 参数可以不同也可以相同。它的有用之处在于，MPI 库实现了通信调度，使程序不再挂起或崩溃。我们之前的代码很复杂，需要检查进程号是奇数还是偶数，现在可以替换为调用 MPI_Sendrecv 函数。如果发送和接收使用的是同一个缓冲区，那么 MPI 库还提供了一个函数：

```
int MPI_Sendrecv_replace(
      void*          buf_p          /* in/out */,
      int            buf_size       /* in     */,
      MPI_Datatype   buf_type       /* in     */,
      int            dest           /* in     */,
      int            send_tag       /* in     */,
      int            source         /* in     */,
      int            recv_tag       /* in     */,
      MPI_Comm       communicator   /* in     */,
      MPI_Status*    status_p       /* in     */);
```

### 3.7.4 并行奇偶交换排序算法的重要内容

134 我们已经设计了如下所示的并行奇偶排序算法：

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

从安全性的角度看，可以使用 MPI_Sendrecv 实现消息的接收和发送：

```
MPI_Sendrecv(my_keys, n/comm_sz, MPI_INT, partner, 0,
      recv_keys, n/comm_sz, MPI_INT, partner, 0, comm,
      MPI_Status_ignore);
```

接下来，就需要确认我们要保留哪些键值了。假设我们保留较小的键值。那么，我们希望在 $2n/p$ 个键值中保存最小的 $n/p$ 个键值。一个显而易见的方法是在列表中用串行算法对这 $2n/p$ 个键值进行排序，然后保留列表的前半段。然而，排序是相对开销比较大的操作，因为我们已经有 2 个有着 $n/p$ 个键值且已排序的列表，那么只需要通过合并它们为一个列表就可以节省很多开销。事实上还可以做得更好，因为我们不需要完全的排序：一旦发现了最小的 $n/p$ 个键值，就可以退出了。具体的实现见程序 3-16。

为了取得最大的 $n/p$ 个键值，我们简单地反转合并的顺序，即从 local_n - 1 开始从后向前地操作数组。最后，还有一处可优化的地方：避免复制数组而只是交换指针（详见习题 3.28）。

表 3-9 是“最终优化”后并行奇偶排序算法的运行时间。可以看到，如果它运行在单核处理器上，它会使用排序局部键值所用的串行算法，即快速排序，而不是奇偶交换排序，后者在单核处理器上的运行时间比前者慢。我们将在习题 3.27 中更深入地研究这些时间。

**表 3-9 并行奇偶排序算法的运行时间（单位：毫秒）**

| 进程 | 键值的数量（单位：千） | | | | |
|---|---|---|---|---|---|
| | 200 | 400 | 800 | 1600 | 3200 |
| 1 | 88 | 190 | 390 | 830 | 1800 |
| 2 | 43 | 91 | 190 | 410 | 860 |
| 4 | 22 | 46 | 96 | 200 | 430 |
| 8 | 12 | 24 | 51 | 110 | 220 |
| 16 | 7.5 | 14 | 29 | 60 | 130 |

**程序3-16　并行奇偶排序算法中的Merge_low函数**

```
void Merge_low(
      int  my_keys[],     /* in/out    */
      int  recv_keys[],   /* in        */
      int  temp_keys[],   /* scratch   */
      int  local_n        /* = n/p, in */) {
   int m_i, r_i, t_i;

   m_i = r_i = t_i = 0;
   while (t_i < local_n) {
      if (my_keys[m_i] <= recv_keys[r_i]) {
         temp_keys[t_i] = my_keys[m_i];
         t_i++; m_i++;
      } else {
         temp_keys[t_i] = recv_keys[r_i];
         t_i++; r_i++;
      }
   }

   for (m_i = 0; m_i < local_n; m_i++)
      my_keys[m_i] = temp_keys[m_i];
}  /* Merge_low */
```

## 3.8　小结

消息传递接口（MPI），是一个可以被C、C++和Fortran程序调用的函数库。许多系统使用mpicc编译MPI程序，并通过mpiexec运行它们。C MPI程序需要包括mpi.h的头文件才能使用MPI库定义的函数和宏。

MPI_Init函数建立MPI程序，它应该最先被调用。若程序不使用argc和argv，可以直接传入NULL。

在MPI中，一个**通信子**是一组进程的集合，该集合中的进程之间可以相互发送消息。MPI程序启动后，MPI创建由所有进程组成的通信子，称为MPI_COMM_WORLD。

许多并行程序使用**单程序多数据流**（SPMD）的方法，通过根据不同的进程号转移到不同的分支语句，使得运行一个程序就能够获得运行多个不同程序的效果。用完MPI后，记得调用MPI_Finalize函数结束程序。

要想从一个MPI进程发送数据给另一个进程，可以调用MPI_Send函数，而MPI_Recv函数是用来接收消息。MPI_Send函数的参数描述了数据的内容和它的目的地，而MPI_Recv函数的参数描述了用于存储接收到数据的缓冲区，以及应从哪接收数据。MPI_Recv是**阻塞的**，即调用MPI_Recv后，直到消息收到（或者发生一个错误）前，该函数不会返回。MPI_ Send的行为则由MPI的实现来定义，它可以阻塞或**缓冲**发送的消息。当它阻塞时，在相应的接收启动前它不会返回；如果缓冲消息，MPI会将其复制到它私有的存储空间，一旦复制完成，MPI_Send就会返回。 135 / 136

编写MPI程序时，非常重要的一点是区分**局部**变量和**全局**变量。局部变量只作用于定义它的那个进程，而全局变量则作用于全部进程。在梯形积分法的程序中，梯形的全部数量$n$是一个全局变量，每个进程区间的左、右端点是局部变量。

大多数串行程序是**确定性的**，意味着我们若用同一个程序运行相同的数据得到的结果是一样的。记住，并行程序一般没有这个特性，如果多个进程独立运行，由于进程控制外的事件，它们可能在不同时间点运行到不同的程序。因此，并行程序是**非确定性的**，同样的输入会得到不同的输出。如果MPI程序中的所有进程都打印输出的结果，那么每次程序运行时打印出来的顺序是不同的。因为这个原因，MPI程序用一个进程（进程0）负责打印结果是很常见的方法。当然，调试时我们一般会忽略这个规则，允许每个进程打印各自的调试信息。

大部分 MPI 实现允许所有的进程将结果打印到 stdout 和 stderr。然而，任意一个 MPI 实现都是只允许一个进程（一般使用 MPI_COMM_WORLD 中的进程 0）读取从 stdin 输入的数据。

**集合通信**不同于只涉及两个进程的 MPI_Send 和 MPI_Recv，它涉及一个通信子中的所有进程。为了区分这两种通信方式，MPI_Send 和 MPI_Recv 常常称为**点对点通信**。

两个常用的集合通信函数是 MPI_Reduce 和 MPI_Allreduce。MPI_Reduce 存储全局操作的结果（如求全局总和）到指定的进程，MPI_Allreduce 将结果存储到通信子中的全部进程中。

有些类似于 MPI_Reduce 的 MPI 函数，可能会将同样的参数传递到输入和输出缓冲区中。这种现象称为**参数别名**，MPI 显式地禁止了输出参数与其他参数相混淆。

我们学习了很多重要的 MPI 集合通信函数：

- MPI_Bcast 从单个进程向同一个通信子中的所有其他进程发送消息，这个函数非常有用，例如，进程 0 从 stdin 读取输入数据，而这些数据需要发送给其他进程。
- MPI_Scatter 在各个进程间分配一个数组的元素。如果数组有 $n$ 个元素，进程数量为 $p$，则第一组的 $n/p$ 个元素发送到进程 0，第二组 $n/p$ 个元素发送到进程 1，以此类推。
- MPI_Gather 是 MPI_Scatter 的“逆操作”。如果每个进程都存储了包含 $m$ 个元素的子
137 数组，那么 MPI_Gather 将收集这些元素到一个特定的进程中，先从进程 0 收集元素，然后是进程 1，以此类推。
- MPI_Allgather 类似于 MPI_Gather，但它收集所有的元素并分发给所有的进程。
- MPI_Barrier 用来同步进程；在同一个通信子中的所有进程调用该函数前，所有调用 MPI_Barrier 的进程都不能返回。

分布式内存系统是没有全局共享内存的，所以在编写 MPI 程序时，如何将数据分割并分配到各个进程上是关键点。对于普通的向量与数组，我们可以使用块划分、循环划分以及块 - 循环划分。如果全局向量或数组有 $n$ 个元素，系统中有 $p$ 个进程，那么**块划分**分配开始的 $n/p$ 个元素给进程 0，接下去的 $n/p$ 个元素分配给进程 1，以此类推。**循环划分**以循环方式分配元素，第一个元素分给进程 0，第二个元素分给进程 1，…，第 $p$ 个元素分给进程 $p-1$，分配完开始的 $p$ 个元素后，再从第一个进程开始分配第二轮的 $p$ 个元素，以此类推。**块 - 循环划分**将元素的块以循环划分方式分配给各个进程。

与只涉及 CPU 和主存的操作相比，发送消息的开销是“昂贵”的。而且，用尽可能少的次数发送尽可能多的消息可以节省开销，所以将多个消息合并为一个消息发送是降低开销的好方法。MPI 为此提供了三种方法：通信函数中的 count 参数、派生数据类型和 MPI_Pack/Unpack 函数。派生数据类型是通过指定的数据类型和它们在内存中的相对位置来描述任意类型集合的数据。在本章中，我们还简单地介绍了如何使用 MPI_Type_create_struct 建立派生数据类型。在习题中，我们将继续探讨其他的方法，并介绍 MPI_Pack/Unpack。

当我们为并行程序计算运行时间时，我们一般关注经过的总时间或者叫“墙上时钟时间”，即运行一段代码所需要的时间，它包括用户级代码、库函数、用户代码调用系统函数的运行时间以及空闲时间。有两种方法来获得该时间：GET_TIME 和 MPI_Wtime。前者是 time.h 定义的宏，可以从本书的网站下载。在串行代码中的使用方法如下：

```
#include "timer.h"  // From the book s website
. . .
double start, finish, elapsed;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
elapsed = finish - start;
138 printf("Elapsed time = %e seconds\n", elapsed);
```

而MPI提供了函数MPI_Wtime来取代GET_TIME。计时并行程序比串行程序复杂得多，在理想情况下，我们想要在代码的开头先同步各个进程，在最慢的进程完成代码后报告时间。MPI_Barrier函数能非常方便地同步进程，调用它的进程会阻塞直到通信子中的所有进程都调用过该函数。使用以下代码模板能够计算MPI程序的运行时间：

```
double start, finisn, loc_elapsed, elapsed;
. . .
MPI_Barrier(comm);
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
loc_elapsed = finish - start;
MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX,
      0, comm);
if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

并行程序计时另一个问题是，多次运行同段代码后的计时结果变化很大，例如，操作系统可能有一个或多个进程空闲从而让其他进程先运行。因此，我们需要多次运行后报告时间最短的结果。

统计完时间后，可以用**加速比**或者**效率**来估算程序的性能。加速比是串行运行时间与并行运行时间之比，而效率是加速比除以进程总数。加速比的理想化时间为$p$，即进程数，而理想的效率是1。我们一般不可能获得这么好的结果，但获得接近的结果还是可能的，尤其是，当$p$较小而问题规模$n$比较大时。**并行开销**是并行程序运行时间的一部分，指的是不能被串行程序执行的额外时间。在MPI程序中，并行开销来自于通信，当$p$很大而$n$较小时，并行开销占据多数的总运行时间是正常的，此时加速比和效率都很低。增加问题的规模（$n$），随着$p$的增加，效率却没有递减，则可以称该并行程序是**可扩展的**。

MPI_Send既可以阻塞也可以缓冲输入，如果一个MPI程序的正确行为取决于MPI_Send正在缓冲的输入，则它是**不安全的**，这经常发生于多个进程第一次调用MPI_Send，然后调用MPI_Recv的情况。如果调用MPI_Send不采用缓冲方式，那么它们会一直阻塞直到相应的MPI_Recv被调用，然而这个情况却永远不会发生，例如，进程0和进程1想要相互发送数据，两者都先发送消息后再接收，进程0会等待进程1调用MPI_Recv，而进程1则一直阻塞在MPI_Send，等待进程0调用MPI_Recv。因此，进程就**死锁**了，双方都阻塞并等待永不发生的事件发生。 [139 ~ 140]

MPI程序能通过互相替换每个MPI_Send和MPI_Recv函数来检查是否安全。MPI_Ssend使用与MPI_Send相同的参数，但它一直阻塞直到对应接收端开启，这个额外的“s”字母代表同步，如果使用MPI_Ssend的MPI程序根据要求的输入和通信规模正确地完成了运行，则该程序为安全的。

一个不安全的MPI程序可以通过多种方法变为安全的，程序员可以调度MPI_Send和MPI_Recv使某些进程（如偶数序号的进程）先调用MPI_Send，而其他进程（如奇数序号的进程）先调用MPI_Recv。另外，可以使用MPI_Sendrecv或者MPI_Sendrecv_replace，这些函数各发送和接收一次消息，它们各自保证程序不会崩溃或死锁。MPI_Sendrecv的发送和接收缓冲使用不同的参数，而MPI_Sendrecv_replace则使用相同的参数。

## 3.9 习题

3.1 在问候程序中，如果strlen(greeting)代替strlen(greeting)+1来计算进程1、2、…、comm_sz-1发送消息的长度，会发生什么情况？如果用MAX_STRING代替strlen(greeting)+1又会是什么结果？你可以解释这些结果吗？

3.2 改变梯形积分法，使其能够在 comm_sz 无法被 $n$ 整除的情况下，正确估算积分值（假设 $n \geqslant$ comm_sz）。

3.3 梯形积分法程序中哪些变量是局部的，哪些是全局的？

3.4 mpi_output.c 程序中，每个进程只打印一行输出。修改程序，使程序能够按进程号的顺序打印，即，进程 0 先输出，然后进程 1，以此类推。

3.5 二叉树中有着从根到每个结点的最短路径，这条路径的长度称为该结点的**深度**。每一个非叶子结点都有 2 个子结点的二叉树叫做**满**二叉树，每个叶子结点深度都相同的满二叉树称为**完全**二叉树。如图 3-14 所示，用数学推导证明：如果 $T$ 是一棵有着 $n$ 个叶子结点的完全二叉树，那么叶子结点的深度为 $\log_2(n)$。

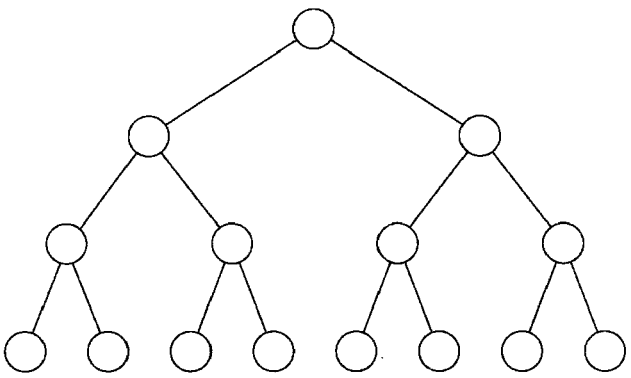


图 3-14 一棵完全二叉树

3.6 假设 comm_sz = 4，$x$ 是一个拥有 $n = 14$ 个元素的向量

a. 如何用块划分法在一个程序的进程间分发 $x$ 的元素。

140

b. 如何用循环划分法在一个程序的进程间分发 $x$ 的元素。

c. 如何用 $b = 2$ 大小的块用块 - 循环划分法在一个程序的进程间分发 $x$ 的值。

你的分配方法应该通用性足够强，无论 comm_sz 和 $n$ 取何值都能分配。你应该使你的分配“公正”，如果 $q$ 和 $r$ 是任意的 2 个进程，分给 $q$ 和 $r$ 的 $x$ 分量个数的差应尽可能小。

3.7 如果通信子只包含一个进程，不同的 MPI 集合通信函数分别会做什么。

3.8 假定 comm_sz = 8，$n = 16$。

a. 画一张图来说明进程 0 要分发 $n$ 个元素的数组，怎样使用拥有 comm_sz 个进程的树形结构的通信来实现 MPI_Scatter。

b. 画一张图来说明原先分布在 comm_sz 个进程间的 $n$ 个数组元素由进程 0 保存，怎样使用树形结构的通信来实现 MPI_Gather。

3.9 编写一个 MPI 程序实现向量与标量相乘以及向量点积的功能。用户需要输入 2 个向量和一个标量，都由进程 0 读入并分配给其他进程，计算结果由进程 0 计算和保存，并最终由进程 0 打印出来。假定向量的秩 $n$ 可以被 comm_sz 整除。

3.10 程序 3.9 的 Read_vector 函数中，实际上使用了 local_n 作为 MPI_Scatter 的 2 个形式参数

141 send_count 和 recv_count，为什么程序仍然能正常运行？

3.11 求 $n$ 个数和的表达式为：

$$x_0 + x_1 + \cdots + x_{n-1}$$

这 $n$ 个数值的前缀和（prefix sum）是 $n$ 个部分和：

$$x_0,\ x_0 + x_1,\ x_0 + x_1 + x_2,\ \cdots,\ x_0 + x_1 + \cdots + x_{n-1}$$

求前缀和事实上是求数值总和的一般化，它不只求出 $n$ 个值的总和。

a. 设计一个串行算法来计算 $n$ 个元素数组的前缀和。

b. 在有 $n$ 个进程的系统上并行化（a）小题中设计的串行程序，每个进程存储一个 x_i 值。

c. 设 $n = 2^k$，$k$ 为正整数。设计一个串行算法，然后将该算法并行化，使得这个并行算法仅需要 $k$ 个通信阶段。

d. MPI 提供一个集合通信函数 MPI_Scan，用来计算前缀和：

```
int MPI_Scan(
      void*          sendbuf_p   /* in  */,
      void*          recvbuf_p   /* out */,
      int            count       /* in  */,
      MPI_Datatype   datatype    /* in  */,
      MPI_Op         op          /* in  */,
      MPI_Comm       comm        /* in  */);
```

该函数对有 count 个元素的数组进行操作，sendbuf_p 和 recvbuf_p 都应该指向有 count 个 datatype 类型元素的数据块。op 参数和 MPI_Reduce 中的 op 一样。编写一个 MPI 程序使每个 MPI 进程生成有 count 个元素的随机数数组，计算其前缀和并打印结果。

3.12 可以用**环形传递**来代替蝶形结构的全归约，在环形传递结构中，如果有 $p$ 个进程，每个进程 $q$ 向进程 $q+1$ 发送数据（进程 $p-1$ 向进程 0 发送数据）。这一过程持续循环直至所有进程都获得理想的结果。我们可以用以下代码实现全归约：

```
sum = temp_val = my_val;
for (i = 1; i < p; i++) {
   MPI_Sendrecv_replace(&temp_val, 1, MPI_INT, dest,
         sendtag, source, recvtag, comm, &status);
   sum += temp_val;
}
```

a. 编写一个 MPI 程序实现这一算法。与蝶形结构的全归约相比，它的性能如何？

b. 修改刚才的程序，实现前缀和的运算。

142

3.13 MPI_Scatter 和 MPI_Gather 存在一些限制，即每个进程必须发送或者接收同样数量的数据。如果不这样的话，就必须改用 MPI_Gatherv 和 MPI_Scatterv 这两个 MPI 函数。查看这些函数的帮助手册，修改你的计算向量和、向量点积程序使其能够处理 $n$ 不被 comm_sz 整除的情况。

3.14 a. 编写一个串行的 C 程序，在主函数中定义一个二维数组，使用以下给出的变量：

```
int two_d[3][4];
```

在主函数中初始化该数组，然后调用函数打印其值。打印函数的原型应该如下所示。

```
void Print_two_d(int two_d[][], int rows, int cols);
```

编写完后请尝试编译程序，你能够解释为什么该程序无法通过编译吗？

b. 参考 C 语言教程（如 Kernighan 和 Ritchie [29]）修改程序后，使其能够顺利编译并运行，它仍然使用一个 2 维的 C 数组。

3.15 我们在 2.2.3 节讨论的二维数组，它的“行优先”存储与 3.4.9 节所提到的以一维数组存储有什么联系？

3.16 假定 comm_sz = 8、向量 $\boldsymbol{x} = (0, 1, 2, \cdots, 15)$，通过块划分方法分配 $\boldsymbol{x}$ 给各个进程，画图表示用蝶形通信结构实现聚集 $\boldsymbol{x}$ 的步骤。

3.17 MPI_Type_contiguous 可以从数组中收集邻接元素，然后创建派生数据类型。它的语法是：

```
int MPI_Type_contiguous(
      int            count        /* in  */,
      MPI_Datatype   old_mpi_t    /* in  */,
      MPI_Datatype*  new_mpi_t_p  /* out */);
```

修改 Read_vector 和 Print_vector 函数，让它们能够使用一种 MPI 的数据类型，这个数据类型是通过调用 MPI_Type_contiguous 生成的，在调用 MPI_Scatter 和 MPI_Gather 函数时，count 参数的值为 1。

3.18 MPI_Type_vector 可以用来将数组中的数据块组合起来构建派生数据类型，这些块大小相同，在数组中的间隔是等距的。它的语法是：

143

```
int MPI_Type_vector(
      int            count         /* in  */,
      int            blocklength   /* in  */,
```

```
int             stride        /* in  */,
MPI_Datatype    old_mpi_t     /* in  */,
MPI_Datatype*   new_mpi_t_p   /* out */);
```

例如，含有18个double类型数据的数组x，我们想建立一个数据类型，对应0、1、6、7、12、13位置的元素，则可以调用：

```
int MPI_Type_vector(3, 2, 6, MPI_DOUBLE, &vect_mpi_t);
```

总共分为3个数据块，每块有2个元素，且每块间的间隔为6个double型元素。

请编写Read_vector和Print_vector函数，使进程0读取和打印以块-循环划分方法分割的向量。但是要注意：不要使用MPI_Scatter或者MPI_Gather函数，用这两个函数操作由MPI_Type_vector创建的类型会引起一个技术性问题（详见[23]）。进程0的Read_vector只是循环地发送数据，进程0的Print_vector只循环地接收数据。其余的进程通过调用一次MPI_Recv和MPI_Send就能完成它们对Read_vector和Print_vector的调用。进程0应该使用由MPI_Type_vector生成的数据类型，其他进程只使用count作为参数传递给通信函数，因为它们正在接收/发送的元素将会被连续地存储在数组中。

3.19 MPI_Type_indexed函数可以用来建立取自任意数组元素的派生数据类型。它的语法为：

```
int MPI_Type_indexed(
      int             count                      /* in  */,
      int             array_of_blocklengths[]    /* in  */,
      int             array_of_displacements[]   /* in  */,
      MPI_Datatype    old_mpi_t                  /* in  */,
      MPI_Datatype*   new_mpi_t_p)               /* out */);
```

与MPI_Type_create_struct不同，MPI_Type_indexed函数中使用old_mpi_t类型，而不是字节来作为偏移量的单位。请用MPI_Type_indexed创建一个派生数据类型，对应于一个矩阵的上三角部分的数据。如4×4矩阵，

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

上三角部分的元素分别为0、1、2、3、5、6、7、10、11、15。进程0以读取一维数组的方式读入$n \times n$矩阵，创建派生的数据类型，并通过调用一次MPI_Send发送矩阵的上三角部分。进程1通过调
144 用MPI_Recv接收这部分数据并打印。

3.20 函数MPI_Pack和MPI_Unpack提供了从分组数据中产生派生数据类型的另一种方法。MPI_Pack每次复制一块要发送的数据到用户提供的缓冲区，该缓冲区既可以接收数据也可以发送数据。当接收到数据后，可以使用MPI_Unpack将接收缓冲区中的数据解包。MPI_Pack的语法为：

```
int MPI_Pack(
      void*           in_buf          /* in     */,
      int             in_buf_count    /* in     */,
      MPI_Datatype    datatype        /* in     */,
      void*           pack_buf        /* out    */,
      int             pack_buf_sz     /* in     */,
      int*            position_p      /* in/out */,
      MPI_Comm        comm            /* in     */);
```

可以用下面的代码打包梯形积分法程序的输入数据：

```
char pack_buf[100];
int position = 0;

MPI_Pack(&a, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&b, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&n, 1, MPI_INT, pack_buf, 100, &position, comm);
```

关键是position参数，当调用MPI_Pack时，该参数应该指向pack_buf中第一个可访问元素的位置；当MPI_Pack返回时，该参数指向pack_buf在数据打包后第一个可以被访问的位置。所以，当进程0执行这段代码时，所有进程可以调用MPI_Bcast：

```
MPI_Bcast(pack_buf, 100, MPI_PACKED, 0, comm);
```

注意，用于表示打包缓冲区的MPI数据类型是MPI_PACKED。现在，除了进程0以外，其他进程都可以用MPI_Unpack来解包数据：

```
int MPI_Unpack(
      void*          pack_buf         /* in     */,
      int            pack_buf_sz      /* in     */,
      int*           position_p       /* in/out */,
      void*          out_buf          /* out    */,
      int            out_buf_count    /* in     */,
      MPI_Datatype   datatype         /* in     */,
      MPI_Comm       comm             /* in     */);
```

MPI_Unpack函数以MPI_Pack相反的步骤进行操作，从position=0的位置一块块解压数据。请为梯形积分法编写另一个Get_input函数，该函数需要在进程0上使用MPI_Pack函数，并在其他进程上使用MPI_Unpack函数。

3.21 你的系统和我们的系统比较起来有哪些差别？你的矩阵-向量乘法程序的运行时间是多少？给定comm_sz和$n$的值后时间上有什么变化？结果集中在最小值、平均值还是中位数附近？ 145

3.22 使用MPI_Reduce执行梯形积分法的程序并统计运行时间。你怎样选择梯形的数量$n$？比较一下最小运行时间、平均运行时间以及运行时间中位数。加速比和效率分别如何？基于你选择的数据，你认为梯形积分法是可扩展的吗？

3.23 尽管我们不知道MPI_Reduce的内部具体实现，但我们猜测它使用了类似二叉树的结构。如果确实如此，我们认为它的运行时间以$\log_2(p)$的速率递增，因为二叉树有大约$\log_2(p)$层（$p$=comm_sz）。串行梯形积分法程序的运行时间与梯形的个数$n$成正比，并行程序简单地为每个进程分配$n/p$个梯形，每个进程采用串行计算的方法处理它们分配到的$n/p$个梯形，然后使用MPI_Reduce将各个进程的计算结果进行归约求总和。我们获得了一个统计并行梯形积分程序的总运行时间公式：

$$T_{并行}(n,p) \approx a * \frac{n}{p} + b\log_2(p)$$

$a$和$b$为常数。

a. 使用上述公式、习题3.22中的计时以及你常用的数值计算程序（如MATLAB），通过最小二乘法估计$a$和$b$的值。

b. 使用上面的公式和（a）小题中估计到的$a$、$b$值来计算运行时间，评价这种估算运行时间方法的准确度。

3.24 看看编程作业3.7，计算发送消息开销的代码应该在count参数为0的情况下也能正常运行。在你系统上，当count=0时，会发生什么情况？你能解释为什么发送0字节的数据仍然会有非0的时间开销吗？

3.25 如果comm_sz=p，我们认为理想的加速比是$p$，有没有可能得到更好的结果？

a. 一个计算向量求和的并行程序，如果只计时向量相加的部分（即不管输入和输出的时间），那么怎样才能使程序获得比$p$更好的加速比？

b. 程序若能够取得高于$p$的加速比，则称为**超线性加速比**。我们的向量求和程序只有克服“资源限制”才能获得超线性加速比。有哪些资源限制？程序有可能在不克服资源限制的情况下获得超线性加速比吗？ 146

3.26 串行的奇偶交换排序算法排序一个$n$元素列表时，所用阶段数远远小于$n$。作为一个极端的例子，如果输入的列表已经完全排好序了，算法只需要0个阶段。

a. 编写一个串行Is_sorted函数来探测输入列表是否已经排序。

b. 修改串行奇偶交换排序算法，使其在每个阶段能够检测列表是否已经排完序。

c. 如果列表中的 $n$ 个元素是随机产生的，那么检查列表是否已经排好序的步骤使得算法中的哪个部分能获得性能改进？

3.27 计算并行奇偶排序算法的加速比和效率，程序能获得线性加速比吗？是可扩展的吗？是强可扩展的还是弱可扩展的？

3.28 修改并行奇偶交换排序算法，使 Merge 函数在发现最小或最大元素后只是简单地交换数组指针，请问这一改变对于整体的运行时间会有怎样的影响？

## 3.10 编程作业

3.1 使用 MPI 实现 2.7.1 节讨论的直方图程序，进程 0 读取输入的数据，并将它们分配到其余进程，最后进程 0 打印该直方图。

3.2 假设我们向一个正方形飞镖板随机地投掷飞镖，飞镖板的边长为 2 英尺，靶心在正中央。再在正方形板上画了一个半径为 1 英尺的圆，面积为 $\pi$ 平方英尺。如果飞镖击中靶子后的得分是平均分布的（我们总能投进正方形区域），则击中圆形区域内的数量应该大致满足等式：

$$\frac{\text{击中圆内的投掷次数}}{\text{全部的投掷次数}} = \frac{\pi}{4}$$

我们可以使用这个公式配合随机数生成器来估计 $\pi$ 的值：

147
```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

这称为蒙特卡洛方法，因为它使用了随机特性（飞镖投掷）。

编写一个用蒙特卡洛方法估计 $\pi$ 的 MPI 程序，进程 0 读入总的投掷次数，并把它们广播给各个进程。使用 MPI_Reduce 求出局部变量 number_in_cycle 的全局总和，并让进程 0 打印它。击中圆内部的次数和投掷总数可能要使用 Long Long int 类型的数值来表示，为了获得较精确的 $\pi$ 估计值，这两个数值应该要大一些。

3.3 编写一个 MPI 程序，采用树形通信结构来计算全局总和。首先计算 comm_sz 是 2 的幂的特殊情况，若能够正确运行，改变该程序使其适用于所有 comm_sz 的值。

3.4 编写一个 MPI 程序，用蝶形通信结构计算全局总和。首先计算 comm_sz 是 2 的幂的特殊情况。你能改变该程序使其适用于任意数目的进程吗？

3.5 实现矩阵与向量相乘的程序，其中矩阵以列为单位分块。进程 0 读取输入的矩阵，然后循环地向其他进程分发数据。假定矩阵是 $n \times n$ 矩阵，$n$ 能够被 comm_sz 整除。你可能需要先查看下 MPI 函数 MPI_Reduce_scatter 的使用方法。

3.6 实现 $n \times n$ 矩阵与向量相乘的程序，其中矩阵以子矩阵块为单位分割。假设向量分布在各个进程中。同样，由进程 0 读取整个矩阵，将矩阵分块为子矩阵后发送给其余进程。假设 comm_sz 是个完全平方数，且 $\sqrt{\text{comm\_sz}}$ 能整除矩阵的阶 $n$。

3.7 ping-pong 是一种通信，数据从进程 A 传送到进程 B（ping）、然后又传送回进程 A（pong）。统计一段间隔内重复 ping-pong 的次数是估计发送消息所需开销的常见方法。在你的系统上用 C 语言的 clock 函数统计一个 ping-pong 程序的运行时间，在 clock 函数给出非 0 的运行时间前，程序需要运行多久？用 clock 函数得到的计时结果与从 MPI_Wtime 函数得出的时间相比，有怎样的区别？

3.8 并行归并排序（merge sort）程序在开始时，会将 $n$/comm_sz 个键值分配给每个进程，程序结束时，所有的键值会按顺序存储在进程 0 中。为了做到这点，它使用了我们在计算全局总和程序中所用到的

树形结构通信模式。当进程接收到另一个进程的键值时，它将该键值合并进自己排完序的键值列表中。编写一个程序实现并行归并排序。进程 0 应该读入 $n$ 的值，将其广播给其余进程。每个进程需要使用随机数生成器来创建 $n/$comm_sz 的局部 int 型数据列表。每个进程先排序各自的局部列表，然 148
后进程 0 收集并打印这些局部列表。然后，这些进程使用树形结构通信合并全局列表给进程 0，并打印最终结果。

3.9　编写一个程序来说明改变数据结构的分配方式所需要的代价。向量的分割方式从块划分变为循环划分，需要花多长时间？反过来呢？ 149

第4章
An Introduction to Parallel Programming

# 用 Pthreads 进行共享内存编程

从程序员角度看，共享内存系统中的任意处理器核都能够访问所有的内存区域（见图 4-1）。因此，协调各个处理器核工作的一个方法，就是把某个内存区域设为“共享”，这是并行编程中常见的方法。确实，你可能会产生疑问：为什么不让所有的并行程序都使用共享内存的编程方法。在本章中，我们会看到由共享内存引起的一些问题，这些问题与在分布式系统中遇到的问题不同。

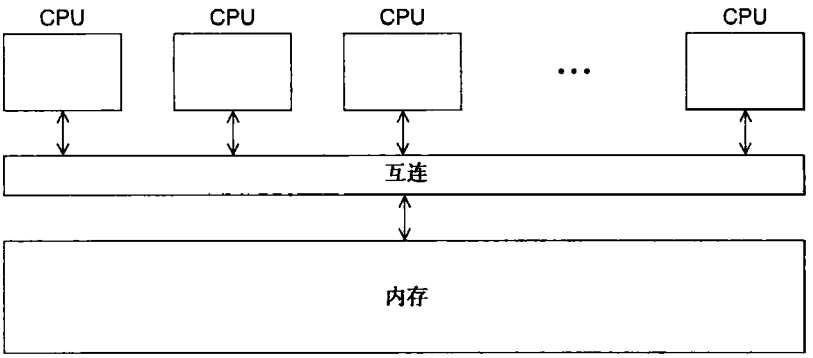



图 4-1　一个共享内存系统

例如，在第 2 章中我们看到，如果不同的处理器核尝试更新共享内存区域上同一位置的数据，那么会导致共享区域的内容无法预测。我们把对共享内存区域进行更新的代码段称为临界区（critical section）。在本章中，我们会看到临界区的一些实例，并学习控制临界区访问的一些方法。

我们还将讨论共享内存编程的其他问题和技巧。在共享内存编程中，运行在一个处理器上的一个程序实例称为**线程**（不同于 MPI，在 MPI 中称为进程）。我们学习如何同步线程，让每个线程在其他线程完成其他工作前等待；我们学习如何让一个线程进入睡眠状态直到某个条件被触发。我们将遇到一些情况，如临界区很大。然而，我们看到，借助某些工具，可以对大型代码段进行颗粒化分析和访问，从而让程序中更多的部分得以并行化。我们看到，使用高速缓存实际上会使共享内存程序的性能降低。最后，我们还看到，在连续的调用之间“维持状态”，可能会导致不一致甚至错误的结果。

本章使用 POSIX® 线程库来实现共享内存的访问。第 5 章介绍另一个共享内存编程方法：OpenMP。

## 4.1　进程、线程和 Pthreads

回忆第 2 章谈到的共享内存编程，线程在一定程度上与 MPI 的进程类似。然而，大体上，它
151 是轻量级的进程。进程是正在运行（或挂起）的程序的一个实例，除了可执行代码外，它还包括：

- 栈段。

- 堆段。
- 系统为进程分配的资源描述符，如文件描述符等。
- 安全信息，如进程允许访问的硬件和软件资源。
- 描述进程状态的信息，如进程是否准备运行或者正在等待某个资源，寄存器中的内容（包括程序计数器数值）等。

在大多数系统中，在默认状态下，一个进程的内存块是私有的：其他进程无法直接访问，除非操作系统进行干涉。这么设计是有意义的。如果正在使用文档编辑器写程序（一个运行文档编辑器的进程），肯定不希望浏览器（另一个进程）覆盖文档编辑器正使用的内存数据。这一设计在多用户环境下更为重要，一个用户的进程是绝对不允许访问其他用户进程拥有的内存的。

然而，这些并不是我们在运行共享内存程序时所需要的。最低限度下，我们希望有些数据对多个进程都是可用的。因此，典型的共享内存“进程”允许了进程间互相访问各自的内存区域。它们也经常共享一些区域，例如共享对 stdout 的访问。事实上，除了它们各自拥有独立的栈和程序计数器外，为了方便，它们基本上可以共享所有其他区域。为了方便管理，一般采用的方法是：启动一个进程，然后由这个进程生成这些“轻量级”进程。**轻量级进程**由此得名。

更通用的术语是**线程**，它来自于“控制线程”的概念，控制线程是程序中的一个语句序列。建议在单个进程中使用术语控制流在共享内存的程序中，一个进程中可以有多个控制线程。 152

我们之前提到，本章使用的是 POSIX 线程库，也经常称为 Pthreads 线程库。POSIX [41] 是一个类 Unix 操作系统（如 Linux、Mac OS X）上的标准库。它定义了很多可以运行于这些系统上的功能，尤其是它定义了一套多线程编程的应用程序编程接口。

Pthreads 不是编程语言（如 C 或 Java），而是与 MPI 一样，它拥有一个可以链接到 C 程序中的库。与 MPI 不同的是，Pthreads 的 API 只有在支持 POSIX 的系统（Linux、Mac OS X、Solaris、HPUX 等）上才有效。与 MPI 不同的是，广泛使用的多线程编程还有很多，如 Java threads、Windows threads、Solaris threads。所有的线程库标准都支持一个基本的概念，即一旦学会了如何使用 Pthreads 进行程序的开发，学习其他线程 API 就很轻松了。

因为 Pthreads 是一个 C 语言库，所以也可以用在C ++ 程序中。然而，标准的C ++ 共享内存线程库（C ++0x）仍在开发中，也许将来在C ++ 程序中，使用这个线程库比使用 Pthreads 库更合适。

## 4.2　“Hello，World”程序

先来看一个 Pthreads 程序。在程序 4-1 中，主函数启动了多个线程，每个线程打印一条消息，然后退出。

### 4.2.1　执行

编译该程序的方法与编译普通的 C 程序是一样的。区别在于，需要链接 Pthreads 线程库：⊖

```
$ gcc -g -Wall -o pth_hello pth_hello.c -lpthread
```

-lpthread 告诉编译器，我们要链接 Pthreads 线程库。注意，是 -lpthread 而不是 -lpthreads。在某些系统上，无需加 -lpthread 选项编译器也会自动链接到该库。

要运行编译好的程序，只要键入

```
$ ./pth_hello <number of threads>
```

153

⊖ 美元符号 $ 是 shell 提示符，这个符号是系统自带的，用户不必输入。同时为了清晰性，假定使用 Gnu 的 C 编译器 gcc，并且一直选择选项 -g、-Wall 和 -o。详见 2.9 节。

程序 4-1 一个 Pthreads "Hello, World" 程序

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  /* Global variable:  accessible to all threads */
6  int thread_count;
7
8  void* Hello(void* rank);  /* Thread function */
9
10 int main(int argc, char* argv[]) {
11    long       thread;  /* Use long in case of a 64-bit system */
12    pthread_t* thread_handles;
13
14    /* Get number of threads from command line */
15    thread_count = strtol(argv[1], NULL, 10);
16
17    thread_handles = malloc (thread_count*sizeof(pthread_t));
18
19    for (thread = 0; thread < thread_count; thread++)
20       pthread_create(&thread_handles[thread], NULL,
21           Hello, (void*) thread);
22
23    printf("Hello from the main thread\n");
24
25    for (thread = 0; thread < thread_count; thread++)
26       pthread_join(thread_handles[thread], NULL);
27
28    free(thread_handles);
29    return 0;
30 }  /* main */
31
32 void* Hello(void* rank) {
33    long my_rank = (long) rank
          /* Use long in case of 64-bit system */
34
35    printf("Hello from thread %ld of %d\n", my_rank,
          thread_count);
36
37    return NULL;
38 }  /* Hello */
```

例如，运行只有一个线程的程序，键入：

```
$ ./pth_hello 1
```

输出类似于：

```
Hello from the main thread
Hello from thread 0 of 1
```

154

运行 4 个线程的程序，键入：

```
$ ./pth_hello 4
```

输出类似于：

```
Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

### 4.2.2 准备工作

我们仔细研究程序 4-1 的代码。首先，我们注意到，这个简单的 C 程序只有 main 函数和另一个函数。程序中包含了我们熟悉的 stdio.h 和 stdlib.h 两个头文件，也有一些新增的不同之处。程序的第 3 行包含了 pthread.h 头文件，这是 Pthreads 线程库的头文件，用来声明

Pthreads 的函数、常量和类型等。

第 6 行定义了一个全局变量 thread_count。在 Pthreads 程序中，全局变量被所有线程所共享，而在函数中声明的局部变量则（通常）由执行该函数的线程所私有。如果多个线程都要运行同一个函数，则每个线程都拥有自己的私有局部变量和函数参数的副本。如果每个线程都有自己私有的栈，那么这种设计是合理的。

需要记住的一点是，全局变量可能会在程序中引发令人困惑的错误。例如，我们写一个有全局变量 int x 的程序，函数 f 内也有一个名为 x 的局部变量，但我们却忘了在函数中声明这个局部变量 x。编译时是不会有错误或者警告出现的，因为函数 f 有权访问全局变量 x。然而在运行时，程序却输出了奇怪的结果，这是由全局变量 x 引起的。过了几天，我们才发现导致全局变量 x 有奇怪值的原因出自函数 f。根据经验法则，应该限制使用全局变量，除了确实需要用到的情况外，比如线程之间共享变量。

程序中的第 15 行表示从命令行中读取需要生成的线程数目。不同于 MPI，Pthreads 程序和普通串行程序一样，是编译完再运行。将命令行参数作为输入值传入程序，由此来确定线程的数目不失为一种简单方便的方法，但也并不仅仅局限于这一种方法。

strtol 函数的功能是将字符串转化为 long int（长整型），它在 stdlib.h 中声明，它的语法形式为：

```
long strtol(
          const char*   number_p   /* in  */,
          char**        end_p      /* out */,
          int           base       /* in  */);
```

155

它返回由 number_p 所指向的字符串转换得到的长整型数，参数 base 是表达这个整数值所用的基（进位计数制）。如果 end_p 不是 NULL，它就指向 number_p 字符串中第一个无效字符（非数值字符）。

### 4.2.3　启动线程

正如我们已经提到的，Pthreads 不同于 MPI 程序，它不是由脚本来启动的，而是直接由可执行程序启动。这样会增加一些复杂度，因为我们需要在程序中添加相应的代码来显式地启动线程，并构造能够存储线程信息的数据结构。

代码第 17 行为每个线程的 pthread_t 对象分配内存，pthread_t 数据结构用来存储线程的专有信息，它由 pthread.h 声明。

pthread_t 对象是一个**不透明**对象。对象中存储的数据都是系统绑定的，用户级代码无法直接访问到里面的数据。Pthreads 标准保证 pthread_t 对象中必须存有足够多的信息，足以让 pthread_t 对象对它所从属的线程进行唯一标识。例如：Pthreads 有一个库函数，利用这个函数可以让线程取得它的专有 pthread_t 对象；还有一个 pthreads 函数，通过检查两个线程 pthread_t 对象来确定它们是否为同一个线程。

在第 19 ~ 21 行的代码中，调用 pthread_create 函数来生成线程。它与大多数的 Pthreads 库函数一样，有一个前缀 pthread，它的语法为：

```
int pthread_create(
      pthread_t*             thread_p                  /* out */,
      const pthread_attr_t*  attr_p                    /* in  */,
      void*                  (*start_routine)(void*)   /* in  */,
      void*                  arg_p                     /* in  */);
```

第一个参数是一个指针，指向对应的 pthread_t 对象。注意，pthread_t 对象不是由 pthread_create 函数分配的，必须在调用 pthread_create 函数前就为 pthread_t 对象分

配内存空间。第二个参数不用，所以只是在函数调用时把 NULL 传递给参数。第三个参数表示该线程将要运行的函数；最后一个参数也是一个指针，指向传给函数 start_routine 的参数。大多数 Pthreads 函数的返回值用于表示线程调用过程中是否有错误。为了减少复杂性，在本章（以及本书后面的内容）中，我们一般忽略 Pthreads 函数的返回值。

接下来，我们仔细研究最后两个参数，由 pthread_create 生成并运行的函数应该有一个类似于下面函数的原型：

156
```
void* thread_function(void* args_p);
```

因为类型 void * 可以转换为 C 语言中任意指针类型，所以 args_p 可以指向一个列表，该列表包含一个或多个 thread_function 函数需要的数值。类似地，thread_function 返回的值也可以是一个包含一个或多个值的列表。在我们的代码中，调用 pthread_create 函数时，传入最后一个参数采用了一个常用的技巧：为每一个线程赋予了唯一的 int 型参数 *rank*，表示线程的编号。首先，我们先解释一下这么做的理由，然后再具体探讨如何做。

考虑以下问题：运行一个生成了两个线程的 Pthreads 程序，当其中一个线程遇到了错误时，我们或者用户如何才能知道是哪个线程出了问题呢？我们不能简单地输出 pthread_t 对象，因为它是不透明的。如果我们启动线程时赋予第一个线程编号为 0，第二个线程编号为 1，那么通过错误信息中线程的编号就能非常容易地判断是哪个线程出错了。

既然线程函数可以接收 void * 类型的参数，我们就可以在 main 函数中为每个线程分配一个 int 类型的整数，并为这些整数赋予不同的数值。当启动线程时，把指向该 int 型参数的指针传递给 pthread_create 函数。然而，程序员会用类型转换来处理此问题：不是在 main 函数中生成 int 型的进程号，而是把循环变量 thread 转化为 void * 类型，然后在线程函数 hello 中，把这个参数的类型转换为 long 型（第 33 行）。

类型转换的结果是“系统定义“的，但大多数 C 编译器允许这么做。不过，如果指针类型的大小和表示进程编号的整数类型不同，在编译时就会收到警告。在我们使用的机器上，指针类型是 64 位，而 int 型是 32 位，为了避免警告，我们用 long 型替代了 int 型。

需要注意的是，我们为每一个线程分配不同的编号只是为了方便使用。事实上，pthread_create 创建线程时没有要求必须传递线程号，也没有要求必须要分配线程号给一个线程。

还需要注意的是，并非由于技术上的原因而规定每个线程都要运行同样的函数。一个线程运行 hello 函数的同时，另一个线程可以运行 goodbye 函数。但与编写 MPI 程序的方法类似，Pthreads 程序也采用“单程序，多数据”的并行模式，即每个线程都执行同样的线程函数，但可以在线程内用条件转移来获得不同线程有不同功能的效果。

### 4.2.4 运行线程

运行 main 函数的线程一般称为**主线程**。所以，在线程启动后，会打印一句：

```
Hello from the main thread
```

同时，调用 pthread_create 所生成的线程也在运行。这些线程通过第 33 行的类型转换代码获得各自的编号，然后打印各自的消息。注意，当线程结束时，由于它的函数的类型有一个返
157 回值，那么线程就应该返回一个值。在本例中，线程没有需要特别返回的值，所以只返回 NULL。

在 Pthreads 中，程序员不直接控制线程在哪个核上运行[⊖]。在 pthread_create 函数中，没有参数用于指定在哪个核上运行线程。线程的调度是由操作系统来控制的。在负载很重的系统上，

⊖ 有些系统（例如，某些 Linux 的实现版本）允许程序员指定线程运行在哪个核上，但这些版本是无法移植的。

所有线程可能都运行在同一个核上。事实上，如果线程个数大于核的个数，就会出现多个线程运行在一个核上。当然，如果某个核处于空闲状态，操作系统就会将一个新线程分配给这个核。

### 4.2.5　停止线程

代码的第 25 行和第 26 行为每个线程调用一次 pthread_join 函数。调用一次 pthread_join 将等待 pthread_t 对象所关联的那个线程结束。pthread_join 的语法为：

```
int pthread_join(
      pthread_t  thread     /* in  */,
      void**     ret_val_p  /* out */);
```

第二个参数可以接收任意由 pthread_t 对象所关联的那个线程产生的返回值。在我们的例子中，每个线程执行 return，最后，主线程调用 pthread_join 等待这些线程完成并最终结束。

将这个函数命名为 pthread_join 的原因是，这个名字常常用于多线程的图解描述。如图 4-2 所示，假设主线程在图中是一条直线，调用 pthread_create 后就创建了主函数的一条分支或派生，多次调用 pthread_create 就会出现多条分支或派生。当 pthread_create 创建的线程结束时，从图 4-2 中可以清楚地看到，这些分支最后又合并（join）到主线程的直线中。

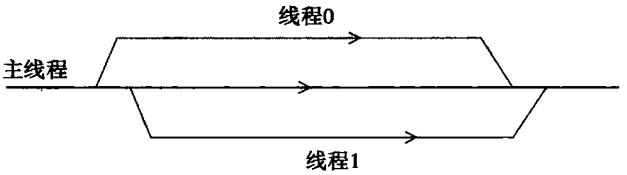



图 4-2　主线程派生与合并两个线程

### 4.2.6　错误检查

为了使程序紧凑易读，我们省略了许多在“真正”程序中常见而重要的细节。例子中的程序（以及很多其他程序）发生错误的很大一部分原因是用户的输入出错或者缺少输入。因此，检查一个程序时，最好一开始先检查命令行参数；允许的话，还可以检查输入的实际线程数目是否合理。如果访问本书的网站，可以下载一份包含基本错误检查的程序。 158

另外，检查由 Pthreads 函数返回的错误代码也是一个好办法，尤其是在你刚开始使用这个库，对具体的函数功能不怎么熟悉的时候。

### 4.2.7　启动线程的其他方法

在我们的例子中，用户通过在命令行键入参数来决定生成多少个线程，然后由主线程来生成这些“辅助”线程。当线程运行时，主线程会打印消息，然后等待其他线程结束。这种多线程的编程方法与 MPI 的编程方法类似，在 MPI 系统中，也会启动一组进程，然后等待它们的完成。

然而，还有一种完全不同的多线程程序设计方法。在这种方法中，辅助线程根据需要而启动。举个例子，一台专门处理旧金山湾区高速公路交通信息的 Web 服务器，假设主线程接收请求，辅助线程完成请求。平常周二的凌晨 1 点，可能只有很少的网络请求，但到了晚上 5 点，却会出现数千个请求。因此，设计 Web 服务器的一个很自然的做法是，请求来到之后，主线程启动辅助线程进行请求处理。

需要知道的是，线程的启动也是有开销的。启动一个线程花费的时间远远比进行一次浮点运算的时间多，所以，“按需启动线程”的方法也许不是使应用程序性能最优化的理想方法。在这种情况下，可能会使用某种比较复杂的模式——一种综合考虑两种方法的模式。主线程可以在程序一开始时就启动所有的线程，当一个线程没有工作可做时，并不结束该线程，而是让该线程处于等待状态，直到再次分配到要执行的任务。在编程作业 4.5 中我们将探讨如何实现这个模式。

## 4.3 矩阵－向量乘法

让我们用 Pthreads 写一个矩阵－向量乘法程序。如果 $\boldsymbol{A}=(a_{ij})$ 是一个 $m\times n$ 的矩阵，$\boldsymbol{x}=(x_0, x_1, \cdots, x_{n-1})^{\mathrm{T}}$ 是一个 $n$ 维列向量[⊖]，矩阵－向量的乘积 $\boldsymbol{Ax}=\boldsymbol{y}$ 是个 $m$ 维的列向量。$\boldsymbol{y}=(y_0, y_1, \cdots, y_{m-1})^{\mathrm{T}}$ 中的第 $i$ 个元素 $y_i$ 是矩阵 $\boldsymbol{A}$ 的第 $i$ 行与 $\boldsymbol{x}$ 的点积：

159

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

图 4-3 矩阵－向量乘法

根据图 4-3，矩阵－向量乘法的串行程序伪代码如下所示。

```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
}
```

通过把工作分配给各个线程将程序并行化。一种分配方法是将线程外层的循环分块，每个线程计算 y 的一部分。例如，假设 $m=n=6$，线程数 thread_count（或 t）为 3，则计算可以按下列情况分配：

| 线程 | y 的一部分 | 线程 | y 的一部分 |
|---|---|---|---|
| 0 | y[0] ,y[1] | 2 | y[4], y[5] |
| 1 | y[2], y[3] | | |

为了计算 y[0]，线程 0 将执行代码：

```
y[0] = 0.0;
for (j = 0; j < n; j++)
   y[0] += A[0][j]* x[j];
```

因此，线程 0 需要访问矩阵 A 的第 0 行以及向量 x 中的每一个元素。更一般地，被分配给 y[i] 的线程将执行代码：

160

```
y[i] = 0.0;
for (j = 0; j < n; j++)
   y[i] += A[i][j]*x[j];
```

因此，该线程需要访问矩阵 A 的第 i 行和向量 x 的所有元素。我们发现，每个线程除了访问各自分配到的矩阵 A 的第 i 行以及 y 分量外，还要访问 x 中的每个元素。这意味着最低限度下，要共

⊖ 记住矩阵和向量的下标从 0 开始。$\boldsymbol{b}$ 是一个矩阵或者向量，而 $\boldsymbol{b}^{\mathrm{T}}$ 代表它的转置。

享向量 x。如果把 A 和 y 都设为共享，看上去好像违反了“只有需要共享的数据才能成为全局变量”的法则。我们将在习题中进一步探讨矩阵 A 和 y 为局部变量时的情况，那时将会发现将它们设置为全局变量是有一定意义的。如果矩阵 A 和 y 是全局变量，主函数就可以简单地通过读取标准输入 stdin 来初始化矩阵 A，乘积向量 y 也可以很容易被主线程打印输出。

做完上述的分析后，我们只要编写每一个线程的代码，确定每个线程计算哪一部分的 y。为了简化代码，假设 $m$ 与 $n$ 都能被 $t$ 整除，在例子中，$m=6$，$t=3$，每个线程能分配到 $m/t$ 行的运算数据，而且，线程 0 处理第一部分的 $m/t$ 行，线程 1 处理第二部分的 $m/t$ 行，以此类推。所以线程 $q$ 处理的矩阵行是：

第一行：$q \times \frac{m}{t}$

最后一行：$(q+1) \times \frac{m}{t} - 1$

有了这些公式，我们就能写出执行矩阵 - 向量相乘的线程函数。见程序 4-2。在这个程序 4-2 中，假设 A、x、y、m 和 n 都是全局共享变量。

**程序 4-2　Pthreads 的矩阵 - 向量乘法**

```
void* Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```

161

如果你已经读过介绍 MPI 的章节，你应该记得，用 MPI 编写一个矩阵 - 向量乘法的程序工作量比较大，因为它的数据结构必须是分布式的，即每个 MPI 的进程只能直接访问自己的局部内存。所以，在 MPI 代码中，需要显式地将 x 分配到每一个进程的内存中。从这个例子看，编写一个共享内存的并行程序比编写分布式内存的程序容易，但我们马上就会看到：共享内存程序也有更复杂的情况。

## 4.4　临界区

因为共享内存区域是较理想的存储访问方式，所以矩阵 - 向量乘法的代码很容易编写。在程序初始化后，线程只读取除了 y 以外的所有变量。即在主函数创建线程后，除了 y 以外，没有任何共享变量被改写。即使是 y，也是每个线程各自改变属于自己运算的那一部分，没有两个或两个以上线程共同处理同一部分 y 的情况。如果情况变了会怎样？如果多个线程需要更新同一内存单元的数据会怎样？我们也在第 2 章和第 5 章中讨论这个问题，所以如果已经阅读过这些章节，你可能已经知道了答案。但我们还是要来看一个例子。

看看这个估算 $\pi$ 值的例子，有很多不同的公式可以用来估算 $\pi$，其中最简单的是：

$$\pi = \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

这不是计算 $\pi$ 的最佳公式，因为等式右边要计算很多项后才能使得结果比较精确，但对我们来

说，需要计算的项越多越好。

该公式的串行运算代码是：

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
   sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

我们尝试用并行化矩阵 - 向量乘法的方法来并行化这个程序：将 for 循环分块后交给各个线程处理，并将 sum 设为全局变量。为了简化计算，假设线程数 thread_count，简称 $t$ 能够整除项目总数 $n$。如果 $\overline{n}=n/t$，那么线程0加上第一部分的 $\overline{n}$ 项，因此，对于线程0，循环变量 i 的范围是
162 $0\sim\overline{n}-1$。线程1第二部分的 $\overline{n}$ 项，循环变量的范围是 $\overline{n}\sim 2n-1$。更一般化地，对于线程 $q$，循环变量的范围是：

$$\overline{qn},\ \overline{qn}+1,\ \overline{qn}+2,\ \cdots,\ (q+1)\ \overline{n}-1$$

而且，第一项，也就是第 $\overline{qn}$ 的符号，当 $\overline{qn}$ 为偶数时，符号为正；当 $\overline{qn}$ 是奇数时，符号为负。线程函数的代码如程序4-3所示。

**程序 4-3　计算 π 的线程函数**

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0)  /* my_first_i is even */
10       factor = 1.0;
11    else  /* my_first_i is odd */
12       factor = -1.0;
13
14    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15       sum += factor/(2*i+1);
16    }
17
18    return NULL;
19 }  /* Thread_sum */
```

如果使用2个线程运行 Pthreads 程序，并且 $n$ 的值也相对较小的话，那么我们发现 Pthreads 程序的结果与串行计算 π 程序的结果差不多。但当 $n$ 值增大时，就会出现一些特别的结果。例如，在双核处理器上我们获得了以下的结果：

| | $n$ | | | |
|---|---|---|---|---|
| | $10^5$ | $10^6$ | $10^7$ | $10^8$ |
| π | 3.141 59 | 3.141 593 | 3.141 592 7 | 3.141 592 65 |
| 1 个线程 | 3.141 58 | 3.141 592 | 3.141 592 6 | 3.141 592 64 |
| 2 个线程 | 3.141 58 | 3.141 480 | 3.141 369 2 | 3.141 646 86 |

可以看到，随着 $n$ 的增加，单线程的估算结果也越来越准确。事实上，$n$ 每增大十倍就能获得更加精确的结果。$n=10^5$ 时，$\pi$ 能精确到小数点后5位；$n=10^6$ 时，精确到小数点后6位等。
163 两个线程的运算结果与 $n=10^5$ 时一样，然而，对于 $n$ 值较大的情况，双线程的计算结果反而变糟。事实上，多次运行这个双线程程序，尽管 $n$ 未变，但每次的结果都不同。现在，对早先我们

提出的问题有了明确的回答："是的，当多个线程尝试更新同一个共享变量时，会出问题。"

让我们看看为什么会引起问题。首先记住，两个变量的相加要使用多条机器指令，例如，用一条 C 语句将存储单元 y 的内容加到存储单元 x 中去：

```
x = x + y;
```

机器的处理过程一般比这个式子更加复杂。因为 x 和 y 中的值都存储在计算机的主存中，无法直接进行加法运算，需要先将它们从主存中加载到 CPU 的寄存器中后，才能进行加法运算。当运算完成后，必须将结果再从寄存器重新存储到主存中。

假设有 2 个线程，每个线程对并存储在自己私有变量 y 中的值进行计算。还假设将这些私有变量加到共享变量 x 中，主线程将 x 的初始值置为 0。每个线程执行以下代码：

```
y = Compute(my_rank);
x = x + y;
```

假设线程 0 计算出的结果为 y = 1，线程 1 计算出的结果为 y = 2，则"正确"结果就应该是 x = 3。但是，可能会出现以下情况：

| 时间 | 线程 0 | 线程 1 |
|---|---|---|
| 1 | 被主线程启动 | |
| 2 | 调用 Compute() | 被主线程启动 |
| 3 | 赋值 y = 1 | 调用 Compute() |
| 4 | 将 x = 0 和 y = 1 放入寄存器 | 赋值 y = 2 |
| 5 | 将 0 和 1 相加 | 将 x = 0 和 y = 2 放入寄存器 |
| 6 | 把 1 存储在存储单元 x | 将 0 和 2 相加 |
| | | 把 2 存储在存储单元 x |

如果在线程 0 存储它的结果前，线程 1 就将 x 的值从内存复制到寄存器，那么线程 0 计算出的结果就会被线程 1 的值重写。问题也可能反过来：如果线程 1 先处理数据，则最后 x 的结果会被线程 0 的值重写。事实上，除非一个线程在其他线程从内存读取 x 前就把它要改写的值写回内存，否则"先到者"的结果肯定会被"后来者"重写。 164

这个例子反映了共享内存编程的一个基本问题：当多个线程尝试更新一个共享资源（在这里是共享变量）时，结果可能是无法预测的。更一般地，当多个线程都要访问共享变量或共享文件这样的共享资源时，如果至少其中一个访问是更新操作，那么这些访问就可能会导致某种错误，我们称之为**竞争条件**（race condition）。在我们的例子中，为了使代码产生正确的结果，需要保证一旦某个线程开始执行 x = x + y，其他线程在它未完成前不能执行此操作。因此，代码 x = x + y 就是一个**临界区**。临界区就是一个更新共享资源的代码段，一次允许只一个线程执行该代码段。

## 4.5　忙等待

当线程 0 要执行 x = x + y 时，它需要先确认线程 1 此时没有在执行同样的语句，一旦线程 0 确认后，它需要通过某些办法，告知线程 1 它目前正在执行该语句，以防止线程 1 在线程 0 的操作完成前，执行该语句而导致出错。最后，当线程 0 完成操作后，也需要通过某些办法，告知线程 1 它已经结束了这个语句的执行，线程 1 此时可以安全地执行这条语句了。

一个不涉及新概念的简单方法就是使用标志变量，设标志 flag 是一个共享的 int 型变量，主线程将其初始化为 0。而且，将下列代码加到我们的例子中：

```
1  y = Compute(my_rank);
2  while (flag != my_rank);
3  x = x + y;
4  flag++;
```

假定线程1先于线程0完成第1行的赋值，当它运行到第2行的while循环时会怎样呢？仔细观察，你会发现这个while是个空循环语句，如果条件flag! =my_rank为真，那么线程1会再一次执行对这个条件的判断。事实上，它会一直执行下去直到条件为假。当检测到条件为假时，才会接下去执行x=x+y这条语句。

因为flag的初始值为0，所以直到线程0执行完flag++前，线程1都无法执行x=x+y。事实上，我们发现，除非线程0发生无法恢复的错误，否则它的进度最终还是会追上线程1。当线程0执行循环时，因为条件为假，所以会执行临界区中的x=x+y，完成后会执行flag++，从而使线程1终于进入临界区。

165

这段代码有一点很关键：*在线程0执行flag++前，线程1不会进入临界区*。如果严格地按照书写顺序来执行代码的话，就意味着直到线程0完成，线程1都不会进入临界区。

while循环语句就是**忙等待**的一个例子，在忙等待中，线程不停地测试某个条件，但实际上，直到某个条件满足之前（在我们的例子中，是flag!=my_rank条件为false），这些测试都是徒劳的。

需要注意的是，“忙等待”这种方法有效的前提是，“严格地按照书写顺序来执行代码”。如果有编译器优化，那么编译器进行的某些代码优化的工作会影响到忙等待的正确执行。编译器无法知道程序是否为多线程的，所以它不知道变量x和flag的值会被其他线程修改。例如，如果我们的代码：

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

只被一个线程执行，那么while(flag != my_rank)和x=x+y语句的执行顺序就不再重要了。编译优化为了充分利用寄存器，可以将某些语句的顺序交换。那么，代码就会变为：

```
y = Compute(my_rank);
x = x + y;
while (flag != my_rank);
flag++;
```

这段代码会使忙等待失效。为了防止出现这种情况，最简单的方法就是关闭编译优化的选项。在习题4.3中我们会见到另一种方法，不需要完全关闭优化选项。

因此，我们发现忙等待不是控制临界区最好的方法。线程1在进入临界区前，只能一遍又一遍地执行flag++，如果线程0由于操作系统的原因出现延迟，那么线程1只会浪费CPU周期，不停地进行循环条件测试。这对性能有极大的影响，另外，关闭编译器优化选项同样也会降低性能。

在继续下一个话题之前，让我们回到程序4-3中的$\pi$计算程序，并将它改写为忙等待代码。第15行是这个函数中的临界区。但是，当线程执行完临界区后，如果它只是简单地将flag的值加1，那么最后flag的值就会比线程数目$t$大，这就会导致所有线程都无法再次进入临界区。当试图再次进入临界区时，各个线程都只能停留在不停地进行循环条件的判断，无法继续后面的操作。所以我们不能简单地对flag值加1，当线程$t-1$离开临界区时，应该将flag值重置为0，需要将flag++的语句改为：

166

```
flag = (flag + 1) % thread_count;
```

**程序 4-4　使用忙等待求全局和的 Pthreads 程序**

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0)
10       factor = 1.0;
11    else
12       factor = -1.0;
13
14    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15       while (flag != my_rank);
16       sum += factor/(2*i+1);
17       flag = (flag+1) % thread_count;
18    }
19
20    return NULL;
21 }  /* Thread_sum */
```

程序 4-4 是修改后的代码，如果编译该程序并开启 2 个线程来运行这个程序，结果是正确的。然而，增加了计算运行时间的代码后，我们发现当 $n = 10^8$ 时，串行求和比并行求和快。在双核系统中，双线程运行这段代码历时 19.5 秒，而串行运算只要 2.8 秒！

为什么会这样？是启动线程和合并线程导致的开销吗？可以编写一个简单的 Pthreads 程序来估算一下：

```
void* Thread_function(void* ignore) {
   return NULL;
}  /* Thread_function */
```

我们发现，在特定的系统上运行上述代码，从启动第 1 个线程到合并第 2 个线程之间所经历的时间少于 0.3 毫秒，所以导致运行慢的原因不在于线程本身的开销。进一步仔细观察这个使用 167
了忙等待的线程函数，我们看到线程的临界区是程序的第 16 行。flag 初始化的值为 0，所以在线程 0 完成临界区运算并将 flag 加 1 之前，线程 1 必须等待。线程 1 开始进入临界区后，线程 0 也需要等待线程 1 完成运算。可见，线程不停地在等待和运行之间切换，显然是等待以及对条件值加 1 的操作使得整体的运行时间增加了 7 倍。

**程序 4-5　循环后用临界区求全局和的函数**

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;
```

```
   return NULL;
}  /* Thread_sum */
```

忙等待不是保护临界区的唯一方法。事实上，还有很多更好的方法。然而，因为临界区中的代码一次只能由一个线程运行，所以无论如何限制访问临界区，都必须串行地执行其中的代码。如果可能的话，我们应该最小化执行临界区的次数。能够大幅度提高性能的一个方法是：给每个线程配置私有变量来存储各自的部分和，然后用 **for** 循环一次性将所有部分和加在一起算出总和。在双核系统上运行程序 4-5，当 $n = 10^8$ 时，总运算时间减为 1.5 秒，有了实质上的改进。

## 4.6 互斥量

因为处于忙等待的线程仍然在持续使用 CPU，所以忙等待不是限制临界区访问的最理想方
168 法。这里，有两个更好的方法：互斥量和信号量。**互斥量**是*互斥锁*的简称，它是一个特殊类型的变量，通过某些特殊类型的函数，互斥量可以用来限制每次只有一个线程能进入临界区。互斥量保证了一个线程独享临界区，其他线程在有线程已经进入该临界区的情况下，不能同时进入。

Pthreads 标准为互斥量提供了一个特殊类型：pthread_mutex_t。在使用 pthread_mutex_t 类型的变量前，必须由系统对其进行初始化，初始化函数如下：

```
int pthread_mutex_init(
      pthread_mutex_t*            mutex_p   /* out */,
      const pthread_mutexattr_t*  attr_p    /* in  */);
```

我们不使用第二个参数，给这个参数赋值 NULL 即可。当一个 Pthreads 程序使用完互斥量后，它应该调用：

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p  /* in/out */);
```

要获得临界区的访问权，线程需调用：

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p  /* in/out */);
```

当线程退出临界区后，它应该调用：

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p  /* in/out */);
```

调用 pthread_mutex_lock 会使线程等待，直到没有其他线程进入临界区；调用 pthread_mutex_unlock 则通知系统该线程已经完成了临界区中代码的执行。

通过声明一个全局的互斥量，可以在求全局和的程序中用互斥量代替忙等待。主线程对互斥量进行初始化，然后，用互斥量替换忙等待和增量标志，在线程进入临界区前调用 pthread_mutex_lock，在执行完临界区中的所有操作后再调用 pthread_mutex_unlock。参见程序 4-6。第一个调用 pthread_mutex_lock 的线程会为临界区“锁门”，其他线程如果也想要进入临界区，也需要先调用 pthread_mutex_lock，这些调用了 pthread_mutex_lock 的线程都会**阻塞**并等待，直到第一个线程离开临界区。所以只有当第一个线程调用了 pthread_mutex_unlock 后，系统才会从那些阻塞的线程中选取一个线程使其进入临界区。这个过程反复执行，直到所有的线程都完成临界区的操作。

对临界区进行“上锁”和“解锁”并不是使用互斥量时唯一的比喻。程序员常常会说，从 pthread_mutex_lock 函数返回的线程“获得了互斥量”或者“拿到了锁”；同样地，调用
169 pthread_mutex_unlock 常称为“释放”一个锁或互斥量。

**程序 4-6　用互斥量计算全局和**

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8     double my_sum = 0.0;
9
10    if (my_first_i % 2 == 0)
11       factor = 1.0;
12    else
13       factor = -1.0;
14
15    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
16       my_sum += factor/(2*i+1);
17    }
18    pthread_mutex_lock(&mutex);
19    sum += my_sum;
20    pthread_mutex_unlock(&mutex);
21
22    return NULL;
23 }  /* Thread_sum */
```

注意，在使用互斥量的多线程程序中，多个线程进入临界区的顺序是随机的，第一个调用 pthread_mutex_lock 的线程率先进入临界区，接下去的线程顺序则由系统负责分配。Pthreads 无法保证线程按其调用 pthread_mutex_lock 的顺序获得进入临界区的锁。但在我们的设定中，只有有限个线程在尝试获得锁的所有权，最终每一个线程都会获得锁。

我们先观察两个没有经过性能优化的程序。这两个程序，一个是采用忙等待计算 π 多线程程序（循环后进入临界区），另一个是互斥量程序。我们发现，只要线程数目不大于核的个数（见表 4-1），这两个程序单线程运行时间与多线程版本的运行时间之比约等于线程数，即

$$\frac{T_{\text{串行}}}{T_{\text{并行}}} \approx \texttt{thread\_count}$$

如果 thread_count 小于或等于核的个数。将 $T_{\text{串行}}/T_{\text{并行}}$ 称为*加速比*，当加速比等于线程数目时，就获得“理想”的性能或*线性加速比*。

比较使用忙等待和互斥量的程序性能，当线程个数少于核的个数时，我们发现两者的执行时间并没有很大差别。对这个发现无需惊讶，因为每个线程只会进入临界区一次，所以除非是临界区的代码很长或者 Pthreads 函数的运行速度慢，否则线程等待进入临界区的时间都不会很长。但是，如果把线程数增加到超过核的个数，那么采用互斥量程序的性能仍然维持不变，但忙等待程序的性能就会下降。 170

**表 4-1　计算 π 的程序，使用 $n=10^8$ 个项目，在一个有两个 4 核处理器的系统上的运行时间（秒）**

| 线程数目 | 忙等待 | 互斥量 |
|---|---|---|
| 1 | 2.90 | 2.90 |
| 2 | 1.45 | 1.45 |
| 4 | 0.73 | 0.73 |
| 8 | 0.38 | 0.38 |
| 16 | 0.50 | 0.38 |
| 32 | 0.80 | 0.40 |
| 64 | 3.56 | 0.38 |

我们可以看到，使用忙等待的多线程程序在线程数超过核的个数时，性能会下降[⊖]。这是合理的。例如，假设有2个核和5个线程，再假设线程0在执行临界区，线程1处于忙等待状态，线程2、线程3、线程4被操作系统挂起。当线程0完成临界区的操作将flag设为1时，线程1进入临界区，操作系统可以调度线程2、线程3、线程4中的一个。假设操作系统调度到线程3，它在while语句上循环等待。当线程1也完成了临界区操作，将flag设为2，操作系统就可以调度线程2或线程4。如果它选择了线程4，则线程3和线程4都会在忙等待中循环，直到操作系统将它们中的一个挂起，并调度线程2。详见表4-2。

## 4.7 生产者－消费者同步和信号量

尽管忙等待总是浪费CPU的资源，但它是我们至今所知的，能事先确定线程执行临界区代码
171 顺序的最适合方法：线程0最先执行，然后线程1，接下来线程2等。如果采用互斥量，那么哪个线程先进入临界区以及此后的顺序由系统随机选取。因为加法计算是可交换的，所以π计算程序的结果不受线程执行顺序的影响。但是，不难想象，仍然会存在一些情况，需要控制线程进入临界区的顺序。例如，假设每个线程生成一个$n \times n$矩阵，然后按照线程号的顺序依次将各个线程的矩阵相乘。但矩阵相乘这种计算是不可交换的，使用互斥量的程序就会出现一些问题：

```
/* n and product_matrix are shared and initialized by the main
thread */
/* product_matrix is initialized to be the identity matrix  */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);
   Free_matrix(&my_mat);
   return NULL;
}  /* Thread_work */
```

**表4-2 采用忙等待，并且线程个数多于核的个数时，可能的线程执行顺序**

| 时间 | flag | 线程 | | | | |
|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | 执行临界区 | 忙等待 | 挂起 | 挂起 | 挂起 |
| 1 | 1 | 终止 | 执行临界区 | 挂起 | 忙等待 | 挂起 |
| 2 | 2 | — | 终止 | 挂起 | 忙等待 | 忙等待 |
| ⋮ | ⋮ | | | | | |
| ? | ? | — | — | 执行临界区 | 挂起 | 忙等待 |

在更复杂的例子中，每个线程都会向其他线程“发送消息”。例如，假设我们有thread_count或$t$个线程，线程0向线程1发送消息，线程1向线程2发消息，…，线程$t-2$向线程$t-1$发消息，线程$t-1$向线程0发送消息。当一个线程“接收”一条消息后，它打印消息并终止。为了实现消息的传递，分配了一个char*类型的共享数组，每个线程初始化消息后，就设定这个共享数组中的指针指向要发送的消息。为了避免引用到没有被定义的指针，主线程将共享数组中的每项都先设为NULL，具体参见程序4-7。当在双核系统上运行多于两个线程的程序时，我们发现消息始终

⊖ 这是典型的运行时间。当使用忙等待，并且线程个数超过核的个数时，运行时间会出现剧烈变化。

未收到。例如，在线程 $t-1$ 把消息复制到 message 数组前，最先运行的线程 0 早已结束。这一 [172]
点也不令人惊奇，如果把第 12 行的 if 语句替换成忙等待的 while 语句，问题就可以得到解决：

```
while (messages[my_rank] == NULL);
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
```

当然，这个方法有着所有使用忙等待程序都有的问题，所以我们更愿意使用其他的方法。

**程序 4-7　使用 Pthreads 发送消息的第一种尝试**

```
1  /* messages has type char**. It's allocated in main. */
2  /* Each entry is set to NULL in main.                */
3  void* Send_msg(void* rank) {
4     long my_rank = (long) rank;
5     long dest = (my_rank + 1) % thread_count;
6     long source = (my_rank + thread_count - 1) % thread_count;
7     char* my_msg = malloc(MSG_MAX*sizeof(char));
8
9     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10    messages[dest] = my_msg;
11
12    if (messages[my_rank] != NULL)
13       printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
14    else
15       printf("Thread %ld > No message from %ld\n", my_rank,
             source);
16
17    return NULL;
18 }  /* Send_msg */
```

在执行完第 10 行的赋值语句后，我们希望“通知”线程号为 dest 的线程，它可以打印消息了，可以把程序改写成：

```
. . .
messages[dest] = my_msg;
Notify thread dest that it can proceed;

Await notification from thread source
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
. . .
```

现在还不太清楚互斥量能否用在这个场合中。我们尝试着通过调用 pthread_mutex_unlock 来“通知”编号为 dest 线程。然而，互斥量初始化后处于开锁状态，因此需要在初始化 message[dest]前增加一个调用给互斥量上锁。但是，问题在于，我们无法知道什么时候线程执行到调用 pthread_mutex_lock。 [173]

为了让我们的表述更清楚，我们让主线程创建并初始化一个互斥量数组，每一个线程对应一个互斥量。然后，我们尝试着做下面的工作：

```
1  . . .
2  pthread_mutex_lock(mutex[dest]);
3  . . .
4  messages[dest] = my_msg;
5  pthread_mutex_unlock(mutex[dest]);
6  . . .
7  pthread_mutex_lock(mutex[my_rank]);
8  printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
9  . . .
```

现在，假设有两个线程，线程 0 运行得远比线程 1 快，所以当它调用第 7 行的 pthread_mutex_lock 时，线程 1 才刚刚运行到第 2 行。线程 0 获得锁，并继续执行 printf 语句，这会导致线程 0 引用空指针（此时，线程 0 的 messages[my_rank]还是 NULL。——译者注），然后使得整个程序崩溃。

还有另外一些方法也可以通过互斥量解决这个问题，参见习题 4.7。然而，POSIX 线程库还

提供另一个控制访问临界区的方法：**信号量**（semaphore）。

信号量可以认为是一种特殊类型的 unsigned int 无符号整型变量，可以赋值为0、1、2、…。大多数情况下，只给它们赋值0和1，这种只有0和1值的信号量称为*二元*信号量。粗略地讲，0对应于上了锁的互斥量，1对应于未上锁的互斥量。要把一个二元信号量用做互斥量时，需要先把信号量的值*初始化*为1，即开锁状态。在要保护的临界区前调用函数 sem_wait，线程执行到 sem_wait 函数时，如果信号量为0，线程就会被阻塞。如果信号量是非0值，就减1后进入临界区。执行完临界区内的操作后，再调用 sem_post 对信号量的值加1，使得在 sem_wait 中阻塞的其他线程能够继续运行。

信号量是由计算机科学家 Edsger Dijkstra 在［13］中首次定义的，它取名自控制铁道换轨的机械设备，用来指定哪列火车能使用轨道。该设备由一个附有标示且能转动的臂轴组成，当臂轴和标杆同向时，列车可以通过；而当臂轴和标杆垂直时，接近的列车必须停止并等待。对应于临界区操作时，臂轴放下表示信号量的值为1；而升起时，则表示信号量值为0。调用 sem_wait 和 sem_post 函数就相当于列车向信号量控制器发送相应的信号。

信号量与互斥量最大的区别在于信号量是没有个体拥有权的，主线程将所有的信号量初始化为0，即“加锁”，其他线程都能对任何信号量调用 sem_post 和 sem_wait 函数。因此，如果
174 使用信号量，Send_msg 函数可以如程序4-8中那样编写。

**程序 4-8　使用信号量让线程发送消息**

```
1  /* messages is allocated and initialized to NULL in main   */
2  /* semaphores is allocated and initialized to 0 (locked) in
        main */
3  void* Send_msg(void* rank) {
4     long my_rank = (long) rank;
5     long dest = (my_rank + 1) % thread_count;
6     char* my_msg = malloc(MSG_MAX*sizeof(char));
7
8     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
9     messages[dest] = my_msg;
10    sem_post(&semaphores[dest])
            /*   Unlock'' the semaphore of dest */
11
12    /* Wait for our semaphore to be unlocked */
13    sem_wait(&semaphores[my_rank]);
14    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
15
16    return NULL;
17 }  /* Send_msg */
```

不同信号量函数的语法为：

```
int sem_init(
      sem_t*     semaphore_p   /* out */,
      int        shared        /* in  */,
      unsigned   initial_val   /* in  */);

int sem_destroy(sem_t*  semaphore_p  /* in/out */);
int sem_post(sem_t*     semaphore_p  /* in/out */);
int sem_wait(sem_t*     semaphore_p  /* in/out */);
```

我们不使用 sem_int：函数的第二个参数，对这个参数只需传入常数0即可。注意，信号量不是Pthreads线程库的一部分，所以需要在使用信号量的程序开头加头文件[⊖]。

⊖ 有些系统，如 Mac OS X，不支持这类的信号量，它们支持一种称为“命名”信号量的信号量，函数 sem_wait 和 sem_post 仍然可用。但 sem_init 要换成 sem_open，而 sem_destroy 则替换为 sem_close 和 sem_unlink。可以在本书网站上找到相关的例子。

```
#include <semaphore.h>
```

最后，需要指出的是：程序 4-8 中的消息传递不涉及临界区。问题已经不再是一段代码一次只能被一个线程执行，而变成了线程 my_rank 在线程 source 发出消息前一直被阻塞。这种一个线程需要等待另一个线程执行某种操作的同步方式，有时称为**生产者 - 消费者同步模型**。 175

## 4.8　路障和条件变量

接下来我们看看共享内存编程的另一个问题：通过保证所有线程在程序中处于同一个位置来同步线程。这个同步点又称为**路障**（barrier），只有所有线程都抵达此路障，线程才能继续运行下去，否则会阻塞在路障处。

路障有很多应用。比如第 2 章讨论的计时多线程程序，希望所有线程能够在同一时间点开始计时，在最后一个线程完成（“最慢”的线程）时再报告时间。具体的代码为：

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish - my_start;

elapsed = Maximum of my_elapsed values;
```

使用这种方法，能够确保所有线程大致在相同的时间点上记录 my_start。

路障另一个非常重要的应用是调试程序。正如你可能已经看到的，当并行程序发生错误时，很难确定具体是哪个位置出现错误。当然，可以让每个线程都打印消息，来表明它运行到程序的哪个点，但当打印的消息越来越多后，这方法就不管用了。路障为调试程序提供了另一种方法：

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

许多 Pthreads 的实现不提供路障，为了使得程序具有可移植性，我们需要自己实现路障。有很多方法来实现路障，我们将具体讨论其中的三种方法。其中的两种方法已经在之前做过介绍，第三种方法使用了新的 Pthreads 对象：条件变量（conditional varialble）。 176

### 4.8.1　忙等待和互斥量

用忙等待和互斥量来实现路障比较直观：我们使用一个由互斥量保护的共享计数器。当计数器的值表明每个线程都已经进入临界区，所有线程就可以离开忙等待的状态了。

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
```

```
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < thread_count);
    . . .
}
```

当然，这种实现会有与其他使用忙等待程序一样的问题：线程处于忙等待循环时浪费了很多CPU周期，并且当程序中的线程数多过于核数时，程序的性能会直线下降。

另一个问题在于共享变量counter。如果我们想要实现第2个路障并重新使用counter这个变量作为计数器，那么会发生什么状况呢？当第一个路障完成时，counter的值为thread_count。除非重置counter的值，否则第一个路障的循环条件counter < thread_count一直处于false状态，路障也就无法阻塞线程了。此外，试图将counter值重置为0几乎注定会失败。如果最后一个线程进入循环并重置counter值，那么其他一些处于忙等待的线程将永远看不到counter = = thread_count的条件成立，并永远处于忙等待中。如果线程在经过路障后重置counter值，另一些线程可能在重置前就已经进入了第二个路障，由这些线程引起的counter增加值就会因重置而丢失，这会导致所有线程都被困在第二个路障的忙等待中。所以，如果想要使用这种实现方式的路障，则有多少个路障就必须要有多少个不同的共享counter变量来进行计数。

### 4.8.2 信号量

我们自然会提出问题：是否能用信号量来实现路障？如果可以，是否能解决采用忙等待和互斥量实现路障的方式里出现的问题？对第一个问题的回答是：当然能。

177
```
/* Shared variables */
int counter;        /* Initialize to 0 */
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier_sem;  /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
}
```

在忙等待的路障中，使用一个计数器来判断有多少线程进入了路障。在这里，我们采用两个信号量：count_sem，用于保护计数器；barrier_sem，用于阻塞已经进入路障的线程。counter_sem信号量初始化为1（开锁状态），第一个到达路障的线程调用sem_wait函数，则随后的线程会被阻塞直到获取访问计数器的权限。当一个线程被允许访问计数器时，它检查counter < thread_count - 1是否成立，如果成立，线程对计数器的值加1并“释放锁”(sem_post(&count_sem))，然后在调用sem_wait(&barrier_sem)后阻塞。另一个方面，若counter = = thread_count - 1，最后一个进入路障的线程重置计数器的值为0，并通过调用sem_post(&count_sem)来“解锁”count_sem。接着，它需要通知所有的线程继续运行，所以它为pthread_count - 1个阻塞在sem_wait(&barrier_sem)的线程分别执行一次sem_

post(&barrier_sem)。

如果出现这种情况：线程提前开始循环执行 sem_post(&barrier_sem)，在其他线程还未调用 sem_wait(&barrier_sem)解锁前，就已经多次调用 sem_post，这种情况是不要紧的。信号量是 unsigned int 类型的变量，调用 sem_post 会对它的值加 1，调用 sem_wait 时只要它的值不为 0 就减 1，当值为 0 时，调用该函数的线程会被阻塞直到信号量的值为正数。所以，在其他线程因调用 sem_wait(&barrier_sem)而阻塞前，循环执行 sem_post(&count_sem)并不会影响程序的正确性，因为最终被阻塞的线程会发现 barrier_sem 的值为正数，然后它们会递减该值并继续运行下去。

应该清楚的是：线程被阻塞在 sem_wait 不会消耗 CPU 周期，所以用信号量实现路障的方法比用忙等待实现的路障性能更佳。那么，如果我们想要执行第二个路障，可以重用第一个路障 178
的数据结构吗？

counter 是可以重用的，因为在所有线程离开路障前，已经小心地重置它了。另外，count_sem 也可以重用，因为线程离开路障前，它已经重置为 1 了。剩下的 barrier_sem，既然一个 sem_post 对应一个 sem_wait，则当线程开始执行第二个路障时，barrier_sem 的值应该为 0。假设有 2 个线程，线程 0 在第一个路障处因调用 sem_wait(&barrier_sem)而阻塞，此时线程 1 正循环执行 sem_post。假设操作系统发现线程 0 处于空闲状态便将其挂起，接着线程 1 继续执行至第二个路障，因为 counter = =0，所以它会执行 else 后面的语句。在递增 counter 值后，它执行 sem_post(&count_sem)，然后执行 sem_wait(&barrier_sem)。

然而，如果线程 0 仍然处于挂起状态，那么它就不会递减 barrier_sem 值，因此当线程 1 抵达 sem_wait(&barrier_sem)时，barrier_sem 的值仍然为 1，它只会简单地将 barrier_sem 减 1 并继续运行下去。这会导致不幸的结果发生：线程 0 被重新调度运行时，会被阻塞在第一个 sem_wait(&barrier_sem)处，而线程 1 在线程 0 进入第二个路障前就已经通过了该路障。可见重用 barrier_sem 导致了一个竞争条件。

### 4.8.3　条件变量

在 Pthreads 中实现路障的更好方法是采用条件变量。**条件变量**是一个数据对象，允许线程在某个特定条件或事件发生前都处于挂起状态。当事件或条件发生时，另一个线程可以通过信号来唤醒挂起的线程。一个条件变量总是与一个互斥量相关联。

条件变量的一般使用方法与下面的伪代码类似：

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

Pthreads 线程库中的条件变量类型为 pthread_cond_t。函数

```
int pthread_cond_signal(pthread_cond_t*  cond_var_p /* in/out */);
```

的作用是解锁一个阻塞的线程，而函数

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p /* in/out */);
```
179

的作用是解锁所有被阻塞的线程。函数

```
int pthread_cond_wait(
      pthread_cond_t*    cond_var_p    /* in/out */,
      pthread_mutex_t*   mutex_p       /* in/out */);
```

的作用是通过互斥量 mutex_p 来阻塞线程，直到其他线程调用 pthread_cond_signal 或者 pthread_cond_broadcast 来解锁它。当线程解锁后，它重新获得互斥量。所以实际上，pthread_cond_wait 相当于按顺序执行了以下的函数：

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

下面的代码是用条件变量实现路障：

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   . . .
}
```

注意，除了调用 pthread_cond_broadcast 函数，其他的某些事件也可能将挂起的线程解锁（Butenhof 的书［6］第 80 页有详细的例子说明）。因此，函数 pthread_cond_wait 一般被放置于 **while** 循环内，如果线程不是被 pthread_cond_broadcast 或 pthread_cond_signal 函数，而是被其他事件解除阻塞，那么能检查到 pthread_cond_wait 函数的返回值不为 0，被解除阻塞的线程还会再次执行该函数。

如果一个线程被唤醒，那么在继续运行后面的代码前最好能检查一下条件是否满足。在我们的例子中，如果调用 pthread_cond_signal 函数从路障中解除阻塞的线程后，在继续运行之前，应该首先查看 counter = =0 是否成立。使用广播唤醒线程尤其需要注意，某些先被唤醒的线程会运行超前并改变竞争条件的状态，如果每个线程在唤醒后都能检查条件，它就能发现条件已经不再满足，然后又进入睡眠状态。

注意，为了路障的正确性，必须调用 pthread_cond_wait 来解锁。如果没有用这个函数对互斥量进行解锁，那么只有一个线程能进入路障，所有其他的线程将阻塞在对 pthread_mutex_lock 的调用上，而第一个进入路障的线程将阻塞在对 pthread_cond_wait 的调用上，从而程序将挂起。

还要注意的是，互斥量的语义要求从 pthread_cond_wait 调用返回后，互斥量要被重新加锁。当从 pthread_mutex_lock 调用中返回，就能“获得”锁。因此，应该在某一时刻通过调用 pthread_mutex_unlock“释放”锁。

与互斥量和信号量一样，条件变量也应该初始化和销毁。对应的函数是

```
int pthread_cond_init(
      pthread_cond_t*            cond_p       /* out */,
      const pthread_condattr_t*  cond_attr_p  /* in  */);

int pthread_cond_destroy(pthread_cond_t*  cond_p  /* in/out */);
```

180
|
181

我们不使用 pthread_cond_init 的第二个参数（调用函数时传递 NULL 作为参数值）。

### 4.8.4 Pthreads 路障

在继续下一个话题之前，我们还要提一下 Open Group，这个正在开发 POSIX 标准的小组，正在为 Pthreads 定义路障接口。然而，正如我们早先提到的，它的使用并不普及，因此我们没有在本书讨论它。可以从习题 4.9 中获取它的 API 细节。

## 4.9 读写锁

现在，让我们看看这类问题：对一个大的、共享的、能够被线程简单搜索和更新的数据结构的访问控制。为了表述清晰，假设共享的数据结构是一个存储 int 型数据的链表，对链表的操作有 Member、Insert 和 Delete。

### 4.9.1 链表函数

链表本身由一组结点组成，每个结点是一个拥有两个成员的结构：一个整型数据和一个指向下一个结点的指针。可以这样定义这个结构：

```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```

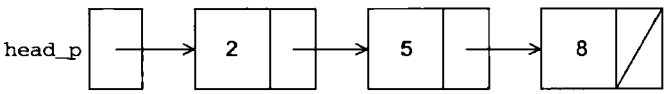



图 4-4　链表

**程序 4-9　Member 函数**

```
1  int  Member(int value, struct list_node_s* head_p)
2     struct list_node_s* curr_p = head_p;
3
4     while (curr_p != NULL && curr_p->data < value)
5        curr_p = curr_p->next;
6
7     if (curr_p == NULL    curr_p->data > value)
8        return 0;
9     else
10       return 1;
11
12    /* Member */
```

一个典型的链表如图 4-4 所示。一个 struct list_node_s * 类型的指针 head_p 指向链表中的第一个结点。最后一个结点的 next 成员是 NULL（在图中用斜杠（/）表示）。

Member 函数（程序 4-9）使用一个指针遍历链表，直到它找到目标值或者知道目标值不在链表中。因为链表是有序的，所以当 curr_p 指针值为 NULL 时，或者当前结点存储的数据成员大于目标值时，就发生后一种情况（目标值不在链表中）。

Insert 函数（程序 4-10）通过查找正确的位置来插入新结点。因为链表是有序的，所以它必须一直查找直到找到一个结点，它的 data 成员大于要插入的 Value。找到这个结点后，将新结点插入到该结点前面。因为链表是单向的，不重新遍历一遍链表就无法“回退”到这个结点之前的位置。有多个方法可以解决这个问题：定义第二个指针 pred_p，指向当前结点的前一个结点。但当我们找到插入的位置并退出查找循环时，pred_p 指向结点的 next 成员需要更新为指向新插入的结点。见图 4-5。

Delete 函数（程序 4-11）与 Insert 函数相似，因为它在查找要删除结点的同时也需要跟

[182] 踪当前结点的前驱结点。前驱结点的 next 成员也需要在查找结束后更新。见图 4-6。

**程序 4-10　Insert 函数**

```
1  int Insert(int value, struct list_node_s** head_p){
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4     struct list_node_s* temp_p;
5
6     while (curr_p != NULL && curr_p->data < value){
7        pred_p = curr_p;
8        curr_p = curr_p->next;
9     }
10
11    if (curr_p == NULL    curr_p->data > value){
12       temp_p = malloc(sizeof(struct list_node_s));
13       temp_p->data = value;
14       temp_p->next = curr_p;
15       if (pred_p == NULL)  /* New first node */
16          *head_p = temp_p;
17       else
18          pred_p->next = temp_p;
19       return 1;
20    } else { /* Value already in list */
21       return 0;
22    }
23 } /* Insert */
```

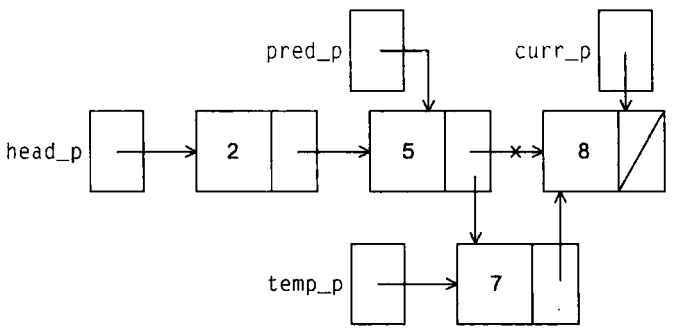



图 4-5　在链表中插入一个新结点

### 4.9.2　多线程链表

现在，让我们试着在一个 Pthreads 程序中使用这些函数。为了对链表共享访问，将 head_p 定义为全局变量。这将简化 Member、Insert 和 Delete 的函数头，因为不需要传递 head_p 或一个指向 head_p 的指针，只需要传递要插入的值。现在，如果有多个线程同时执行这三个函数，结果会是什么？

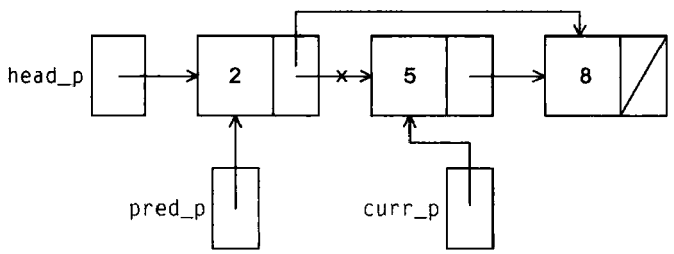



图 4-6　从链表中删除一个结点

**程序 4-11　Delete 函数**

```
1  int Delete(int value, struct list_node_s** head_p){
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4
5     while (curr_p != NULL && curr_p->data < value){
6        pred_p = curr_p;
7        curr_p = curr_p->next;
8     }
9
10    if (curr_p != NULL && curr_p->data == value){
11       if (pred_p == NULL)   /* Deleting first node in list */
12          *head_p = curr_p->next;
13          free(curr_p);
14       } else{
15          pred_p->next = curr_p->next;
16          free(curr_p);
17       }
18       return 1;
19    }else{   /* Value isn't in list */
20       return 0;
21
22 } /* Delete */
```

多个线程可以没有冲突地同时读取一个内存单元，因此很清楚多个线程能够同时执行 Member 函数。另一方面，Delete 和 Insert 函数需要写内存，如果试图让这类操作与其他操作同时执行，那么可能会有问题。例如，假设线程 0 正在执行 Member (5)，与此同时，线程 1 在执行 Delete (5)。链表的当前状态如图 4-7 所示。一个明显的问题是：如果线程 0 正在执行 Member (5)，它将报告 5 在链表中，然而事实上，5 可能在线程 0 返回前就被删除了。第二个明显的问题是：如果线程 0 正在执行 Member(8)，线程 1 可能会在线程 0 查找到存储 8 的结点前先释放存储 5 的结点。尽管典型的 free 实现不覆盖释放的内存，但如果内存在线程 0 到达之前进行了重新分配，那么将会产生严重的问题。例如，如果内存被重新分配用于其他用途，而不是作为链表的结点，那么线程 0 会“认为” next 成员已经设置为垃圾，并且在执行

```
curr_p = curr_p->next;
```

后，对 curr_p 的引用可能会导致段违规。

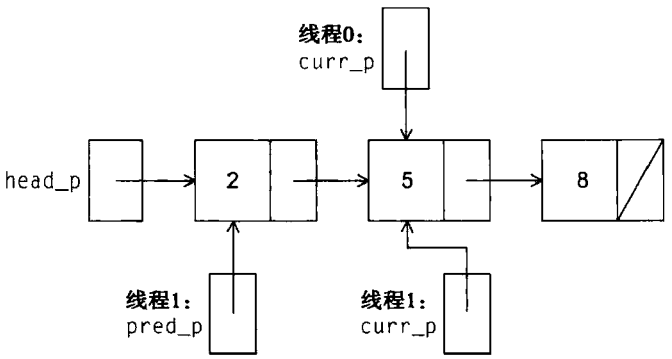



图 4-7　被两个线程同时访问的链表

通常，可以把问题归结为：在执行 Insert 或 Delete 的同时执行其他操作。多线程同时执行 Member 操作，即读链表结点是没有问题的；但当多个线程同时访问链表且至少有一个线程正在执行 Insert 或 Delete 操作，即写链表结点时，是不安全的（见习题 4.11）。

怎么处理这个问题呢？在对链表进行访问前先加锁是一个显而易见的方案。例如，调用这三

个函数时需要用互斥量来保护，所以可以执行以下代码来代替简单的 Member 调用：

```
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```

这个方案也带来显而易见的问题：必须串行访问链表。如果对链表大部分的访问操作是 Member，那么就失去了开发并行性的机会。另一方面，如果对链表大部分的访问操作是 Insert 和 Delete，这可能是最好的解决方案，因为大部分的操作都需要串行执行，而这个解决方案很容易实现。

183 另一个可选择的方案是“细粒度”锁。可以对链表上的单个结点上锁，而不是对整个链表上锁。例如，可以在链表结点的结构中添加一个互斥量：

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```

现在，每次访问一个结点时，必须先对该结点加锁。注意，这要求有一个与 head_p 指针相关联的互斥量。因此，可以像程序 4-12 所示例的那样实现 Member 函数。这个实现比原来的 Member 函数实现更复杂。它还比原来的实现慢，因为每次访问一个结点时，都必须对结点加锁和解锁，所以每一次的结点访问至少增加两次函数调用；另外，如果线程需要等待锁，又会增加附加的延迟。更进一步会产生的问题是，每个结点都增加了一个互斥量域，这必然增加了整个链表对存储量的需求。但另一方面，细粒度的锁可能真的是一个更接近需求的方案。因为只对当前感兴趣的结点加锁，所以多个线程能同时访问链表的不同部分，不管它们在执行什么操作。

185

186

**程序 4-12　用每个链表结点拥有一个互斥量的方法来实现 Member 函数**

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

### 4.9.3　Pthreads 读写锁

前面我们介绍了两种实现多线程链表的方法，这两种方法都不让正在执行 Member 函数的线

程还可以同时访问链表的任意结点。第一个方法在任一时刻只允许一个线程访问整个链表，第二个方法在任一时刻只允许一个线程访问任一给定结点。Pthreads 的**读写锁**可以作为另一种可选方案。除了提供两个锁函数以外，读写锁基本上与互斥量差不多。这两个函数，第一个为读操作对读写锁进行加锁，第二个为写操作对读写锁进行加锁。多个线程能通过调用读锁函数而同时获得锁，但只有一个线程能通过写锁函数获得锁。因此，如果任何线程拥有了读锁，则任何请求写锁的线程将阻塞在写锁函数的调用上。而且，如果任何线程拥有了写锁，则任何想获取读或写锁的线程将阻塞在它们对应的锁函数上。

使用 Pthreads 的读写锁，能用下列的代码保护链表函数（忽略函数返回值）：

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

这个新的 Pthreads 函数的语法是：

```
int pthread_rwlock_rdlock(pthread_rwlock_t*  rwlock_p /* in/out */);
int pthread_rwlock_wrlock(pthread_rwlock_t*  rwlock_p /* in/out */);
int pthread_rwlock_unlock(pthread_rwlock_t*  rwlock_p /* in/out */);
```

就像它们的名字所表示的一样，第一个函数为读加锁，第二个为写加锁，最后一个用于解锁。 187

与互斥量一样，读写锁在使用前应该初始化，在使用后应该销毁。下面的函数用来初始化：

```
int pthread_rwlock_init(
      pthread_rwlock_t*            rwlock_p  /* out */,
      const pthread_rwlockattr_t*  attr_p    /* in  */);
```

与互斥量一样，不使用第二个参数，调用时将传递 NULL 值。下面的函数用来释放一个读写锁：

```
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p /* in/out */);
```

### 4.9.4　不同实现方案的性能

当然，我们想知道这三个实现哪个是“最好”的，因此我们编写了一个小程序，其中包括这三种实现。在程序中，主线程首先向空链表中插入用户指定数量的随机生成的键值。被主线程启动后，每个线程对链表执行用户指定数量的操作。用户还要指定每类操作（Member、Insert、Delete）所占的百分比。然而，哪个操作发生以及对哪个键值进行操作取决于一个随机数生成器。例如，用户可能会指定：插入 1000 个键值到初始的空链表中，线程总共执行 100 000 个操作。而且，她可能还会指定 80% 的操作是 Member，15% 是 Insert，剩下的 5% 是 Delete。然而，由于操作是随机生成的，所以线程可能执行了 79 000 个 Member 调用，15 500 个 Insert 调用，5500 个 Delete 调用。

表 4-3 和表 4-4 列出了对一个初始包含 1000 个键值的链表执行 100 000 个操作所花的时间（单位为秒）。两组数据都在一个拥有 4 个双核处理器的系统上测得。

表 4-3 说明了当 99.9% 的操作是 Member，剩下的 0.1% 被 Insert 和 Delete 操作均分时程序的执行时间。表 4-4 说明了 80% 的操作是 Member，10% 是 Insert，10% 是 Delete 操作所花的时间。我们注意到：在两个表中，当线程数为 1 时，读写锁与单互斥量实现的执行时间是一样 188
的。这很好理解：因为没有对读写锁和互斥量的竞争，这些操作是串行执行的，这两种实现的开

销都包含对链表进行操作前的函数调用和操作后的函数调用。另一方面，使用每个结点一个锁的实现比较慢。这也是好理解的，因为每次单个结点被访问，将会有 2 个操作（一次加锁、一次解锁），因此开销更大。

**表 4-3　链表操作时间：1000 个初始键值，100 000 个操作，99.9% Member，0.05% Insert，0.05% Delete**

| 实现 | 线程数目 | | | |
|---|---|---|---|---|
| | 1 | 2 | 4 | 8 |
| 读写锁 | 0.213 | 0.123 | 0.098 | 0.115 |
| 整个链表一个互斥量 | 0.211 | 0.450 | 0.385 | 0.457 |
| 每个结点一个互斥量 | 1.680 | 5.700 | 3.450 | 2.700 |

**表 4-4　链表操作时间：1000 个初始键值，100 000 个操作，80% Member，10% Insert，10% Delete**

| 实现 | 线程数目 | | | |
|---|---|---|---|---|
| | 1 | 2 | 4 | 8 |
| 读写锁 | 2.48 | 4.97 | 4.69 | 4.71 |
| 整个链表一个互斥量 | 2.50 | 5.13 | 5.04 | 5.11 |
| 每个结点一个互斥量 | 12.00 | 29.60 | 17.00 | 12.00 |

在多个线程的情况下，每个结点一个互斥量的实现仍然维持其低效性。与其他两个实现相比，过多的加锁和解锁使这个实现开销太大。

两个表格最重要的不同之处也许在于：在多线程情况下，读写锁和单互斥量实现的相对性能。当 Insert 和 Delete 操作十分少时，读写锁远好于单互斥量实现。因为单互斥量实现串行化所有的操作，这意味着，如果 Insert 和 Delete 十分少时，读写锁在对链表的并发访问上做得很好。另一方面，如果有相对多的 Insert 和 Delete 时（例如，大约每个 10%），读写锁和单互斥量实现的性能几乎没有差别。因此对链表操作来说，读写锁能够提供很大的性能提升，但只有在 Insert 和 Delete 操作十分少的时候才行。

我们还注意到：如果使用一个互斥量或者每个结点一个互斥量，那么程序在多线程下运行总是比只运行一个线程时更快或者一样快。而且，当 insert 和 delete 的数量相对较大时，读写锁的多线程程序也比单线程快。这对于单互斥量实现来说并不奇怪，因为对链表的有效访问是串
189 行的。对于读写锁实现，当有大量写锁操作时，会出现太多的锁竞争，综合性能将下降得很快。

总之，读写锁实现比单互斥量和每个结点一个互斥量的实现更好。除非 insert 和 delete 操作所占的比例很小，否则串行实现将是更好的方案。

### 4.9.5　实现读写锁

原来的 Pthreads 标准并不包含读写锁，因此一些早期描述 Pthreads 的文本包括了读写锁的实现（例如，见 [6]）。典型的读写锁实现[⊖]定义了一个数据结构，该结构使用两个条件变量（一个对应“读者”，另一个对应“写者”）和一个互斥量。这个数据结构还包含一些成员，分别用于表示：

（1）多少读者拥有锁，即有多少线程同时在读。

---

⊖ 根据 Butenhof 所提出的实现方法的基本大纲。

（2）多少读者正在等待获取锁。

（3）是否有一个写者拥有锁。

（4）多少写者正在等待获取锁。

互斥量用于保护读写锁的数据结构：无论何时一个线程调用其中的任意一个函数（读锁、写锁、解锁），它必须首先锁互斥量，并且无论何时一个线程完成了这些函数调用中的一个，它必须解锁互斥量。在获取互斥量后，线程检查合适的数据成员来决定接下来干什么。例如，如果它想要进行读访问，就检查是否有一个写者当前拥有锁。如果没有，它对活动读者（即同时读的线程）的数量加 1，然后继续执行随后的操作。如果有一个活动写者（有一个写者拥有锁，正在写），就为等待获取锁的读者的数量加 1，并且在读者条件变量上启动一个条件等待。当它被条件唤醒后，它将正在等待的读者的数量减 1，对活动读者的数量加 1，并继续执行随后的操作。写锁函数的实现与读锁函数相类似。

解锁函数的操作取决于线程是一个读者还是一个写者。如果线程是一个读者，且没有其他的活动读者，并且一个写者正在等待，那么它就在返回前发送信号通知写者，使写者继续后继的操作。另一方面，如果线程是写者，则可能同时有读者和写者正在等待，因此线程需要决定它倾向于读者还是写者。因为写者必须互斥访问，很可能写者更难获得锁。因此，许多实现给予写者优先权。编程作业 4.6 进一步探讨了这方面的内容。

## 4.10　缓存、缓存一致性和伪共享[⊖]

多年来，处理器的执行速度比访问主存中数据的速度快得多。如果一个处理器每次操作必须 190
从主存中读数据，那么它将花费大量的时间等待数据从内存中取出后再到达处理器。为了处理这个问题，芯片设计人员已经为处理器增加了相对快速的内存。这个更快的内存就是**缓存**（cache memory）。

缓存的设计考虑了**时间和空间局部性**原理：如果一个处理器在时间 $t$ 访问内存位置 $x$，那么很可能它在一个接近 $t$ 的时间访问接近 $x$ 的内存位置。如果一个处理器需要访问主存位置 $x$，那么就不只是将 $x$ 的内容传入（出）主存，而是将一块包含 $x$ 的内存块传入（出）主存。我们将这样一块内存称为**缓存行**或者**缓存块**。

在 2.3.4 节中，我们已经看到缓存的使用会对共享内存有很大的影响。这是为什么？首先，考虑下列情形：假设 x 是一个共享变量，值为 5，线程 0 和 1 将 x 从内存读入它们各自的缓存，因为它们都想要执行语句：

```
my_y = x;
```

这里，my_y 是一个被两个线程定义的私有变量。现在假设线程 0 执行语句：

```
x++;
```

最后，假设线程 1 正在执行：

```
my_z = x;
```

my_z 是另一个私有变量。

my_z 的值是多少呢？是 5 吗？或者是 6 吗？问题在于：x（至少）有 3 个副本，一个在主存中，一个在线程 0 的缓存中，一个在线程 1 的缓存中。当线程 0 执行 x++时，主存和线程 1 缓存中的值会发生什么变化？这是缓存一致性问题，我们在第 2 章讨论过。大部分系统实现会让缓存感知到它缓存的数据有否变化。在线程 0 执行 x++后，线程 1 中 x 的缓存行将被标记为无效，在赋值 my_z = x 前，运行线程 1 的核就能看到它所存储的 x 的副本已经过期了。这样，运行线

⊖ 这部分内容在第 5 章也有讨论，如果你已经读了那一章，可以跳过这部分。

程 0 的核不得不更新 x 在主存中的副本，然后运行线程 1 的核再从主存得到 x 的更新值。更多的细节见第 2 章。

缓存一致性可能会对共享内存系统的性能有巨大的影响。为了说明这一点，我们再来看看 Pthreads 矩阵 - 向量乘法的例子：主线程初始化一个 $m \times n$ 的矩阵 $A$，以及一个 $n$ 维的向量 $x$。每个线程负责计算乘积向量 $y = Ax$ 的 $m/t$ 个部分。（$t$ 是线程数。）$A$、$x$、$y$、$m$ 和 $n$ 的数据结构都是
[191] 共享的。为了易于参考，我们在程序 4-13 中重写了代码。

如果 $T_{串行}$ 是串行程序执行的时间，$T_{并行}$ 是并行程序执行的时间，并行程序的效率 $E$ 是加速比 $S$ 除以线程数：

$$E = \frac{S}{t} = \frac{\left(\frac{T_{串行}}{T_{并行}}\right)}{t} = \frac{T_{串行}}{t \times T_{并行}}$$

因为 $S \leqslant t$，所以 $E \leqslant 1$。表 4-5 列举了对于不同的数据集和不同的线程数，矩阵 - 向量乘法程序的运行时间和效率。

在每一个样例中，浮点数加法和乘法的总数是 64 000 000，所以如果只考虑算术运算，可以预测到：采用单线程来计算这 3 种输入所花费的时间相差不多。然而，很明显地，情况并不是这样。单线程运行时，8 000 000 × 8 的系统比 8000 × 8000 的系统需要多 14% 的时间，8 × 8 000 000 的系统比 8000 × 8000 的系统需要多 28% 的时间。这个差别部分归因于缓存的性能。

**程序 4-13　Pthreads 矩阵 - 向量乘法程序**

```
1  void *Pth_mat_vect(void* rank) {
2     long my_rank = (long) rank;
3     int i, j;
4     int local_m = m/thread_count;
5     int my_first_row = my_rank*local_m;
6     int my_last_row = (my_rank+1)*local_m - 1;
7
8     for (i = my_first_row; i <= my_last_row; i++) {
9        y[i] = 0.0;
10       for (j = 0; j < n; j++)
11           y[i] += A[i][j]*x[j];
12    }
13
14    return NULL;
15 }  /* Pth_mat_vect */
```

**表 4-5　矩阵 - 向量乘法的运行时间和效率（时间以秒为单位）**

| 线程数 | 矩阵的维数 | | | | | |
|---|---|---|---|---|---|---|
| | 8 000 000 × 8 | | 8000 × 8000 | | 8 × 8 000 000 | |
| | 时间 | 效率 | 时间 | 效率 | 时间 | 效率 |
| 1 | 0.393 | 1.000 | 0.345 | 1.000 | 0.441 | 1.000 |
| 2 | 0.217 | 0.906 | 0.188 | 0.918 | 0.300 | 0.735 |
| 4 | 0.139 | 0.707 | 0.115 | 0.750 | 0.388 | 0.290 |

[192] 当一个核试图更新一个不在缓存中的变量时，就会发生写缺失（write-miss），处理器必须访问主存。缓存分析器（例如 Valgrind [49]）发现当程序运行 8 000 000 × 8 输入时，比其他 2 种输入发生更多的缓存写缺失。大部分写缺失发生在第 9 行。因为结果向量 y 的元素数量远远大于其他两个输入的结果向量元素（8 000 000 与 8000 或 8），而且每个元素必须被初始化，因此这一行减慢了 8 000 000 × 8 个输入程序的执行并不令人惊讶。

当一个核试图访问一个不在缓存中的变量时，就会发生读缺失（read-miss），处理器必须访问主存。缓存分析器发现当程序运行 8 × 8 000 000 输入时，比其他 2 种输入会发生更多的读缺失。这些失效都发生在第 11 行，对这个程序的深入研究（见习题 4.15）表明：不同之处主要在于对 x 的读。这同样不令人惊讶，对这个输入，x 需要读入 8 000 000 个元素，而另外两种输入的 x 只有 8000 或 8 个元素需要读入。

需要注意的是，还会有其他的因素影响单线程程序在不同输入情况下的相对性能。例如，我们还没有考虑虚拟内存（见 2.2.4 节）是否会影响程序不同输入的性能。CPU 访问主存中页表的频率是多少？

我们更感兴趣的是，当线程数增加时，程序执行的效率有很大的不同。输入为 8 × 8 000 000 的双线程程序的效率比输入为 8 000 000 × 8 和 8000 × 8000 的程序效率几乎低 20%。输入为 8 × 8 000 000 的四线程程序比输入为 8 000 000 × 8 的程序效率低约 60%，比输入为 8000 × 8000 的程序的效率低超过 60%。我们还注意到输入为 8 × 8 000 000 的单线程程序也是最慢的，因此，效率公式中的分子也更大：

$$\text{并行效率} = \frac{\text{串行运行时间}}{\text{线程数} \times \text{并行运行时间}}$$

为什么输入为 8 × 8 000 000 的多线程程序的性能差那么多？

答案是：缓存的原因。我们来看一下四线程执行的情况。输入为 8 000 000 × 8 时，y 有 8 000 000 个元素，每个线程分配到 2 000 000 个元素。输入为 8000 × 8000 时，每个线程分配到 2000 个 y 的元素，输入为 8 × 8 000 000 时，每个线程分配到 2 个元素。在我们使用的系统上，一个缓存行是 64 个字节，y 的类型是双精度浮点数，为 8 个字节，一个缓存行能存储 8 个双精度浮点数。

缓存一致性是“行级”的。也就是说，每次缓存行中的任何一个值被改写了，如果该行也存在另一个处理器的缓存中，不只是被写的那个值，在那个处理器上的整个缓存行都会无效。我们使用的系统有 2 个双核处理器，每个处理器有它自己的缓存。假设现在将线程 0 和线程 1 分配给 193
一个处理器，线程 2 和 3 分配给另一个处理器。同时假设对于 8 × 8 000 000 的问题，所有的 y 都存在一个缓存行中。每一次写 y 的某个元素，就会让另一个处理器中的缓存行无效。例如，每次线程 0 用以下语句更新 y[0]：

```
y[i] += A[i][j] * x[j];
```

如果线程 2 或线程 3 也执行这个代码（用于更新 y 的其他元素），它就不得不重新加载 y。每个线程要更新每一个元素，共更新 8 000 000 次。我们看到，在这种线程分配和缓存行分配的配置下，所有线程都不得不重新加载 y 许多次。这种情况在一个元素只被一个线程访问的情况下也会发生，例如，只有线程 0 访问 y[0]。

每个线程都会对分配给它的 y 元素进行更新，两个线程总共更新 16 000 000 次。尽管这些更新的大部分不会迫使线程访问主存，但其中仍然会有很多次访问主存。这称为**伪共享**（false sharing）。假设 2 个拥有各自缓存的线程访问属于同一缓存行的不同变量。再进一步假设至少有一个线程更新了它的变量，那么即使没有线程写另一个线程正在使用的变量，缓存控制器仍然会使整个缓存行无效并强制线程从内存获取变量的值。线程并不共享任何东西（除了一个缓存行），但线程对内存访问的行为好像它们正在共享一个变量，因此把这种现象命名为*伪共享*。

为什么伪共享对其他情况的输入就不是问题？我们来看一下输入为 8000 × 8000 会发生什么。假设将线程 2 分配给一个处理器，线程 3 分配给另外一个处理器（实际上，我们并不知道哪些线程分配给哪些处理器，具体参见习题 4.16）。线程 2 负责计算：

```
y[4000],y[4001],…,y[5999]
```

线程3负责计算：

y[6000],y[6001],…,y[7999]

如果一个缓存行包含8个连续的双精度浮点数，唯一可能会发生伪共享的地方是两个线程各自分配到的值之间的接口。例如，一个缓存行如果包含：

y[5996],y[5997],y[5998],y[5999],y[6000],y[6001],y[6002],y[6003]

伪共享可能会发生在这个缓存行上。然而，线程2在它的 for i 循环的**最后**访问的是：

y[5996],y[5997],y[5998],y[5999]

线程3在 for i 循环的*开始*访问：

194 y[6000],y[6001],y[6002],y[6003]

因此很可能当线程2访问 y[5996]时，线程3已经完成了所有对这四个的访问：

y[6000],y[6001],y[6002],y[6003]

类似地，当线程3访问 y[6003]时，很可能线程2还没有开始访问：

y[5996],y[5997],y[5998],y[5999]

因此，在输入为 8000 × 8000 时，y 元素的伪共享不会是重大问题。类似地，在输入为 8 000 000 × 8 时，y 的伪共享也不太可能会引起问题。我们还注意到：不必太担心 A 或者 x 的伪共享，因为它们的值从来不会被矩阵 - 向量相乘的代码更新。

这也带来了问题：怎样才能在矩阵 - 向量乘法代码中避免伪共享。一个可能的解决方案是用假的元素“填充”y 向量，从而保证一个线程的更新不会影响另一个线程的缓存行。另一个替代方案是每个线程在乘法循环时使用它自己的私有存储器，然后在完成后更新共享存储器。见习题4.18。

## 4.11 线程安全性[⊖]

接下来我们来了解在共享内存编程中另一个可能会发生的问题：*线程安全性*。如果一个代码块能够被多个线程同时执行而不引起问题，那么它是**线程安全的**。

例如，假设要使用多个线程来对一个文件进行“分词”。假设文件由普通的英语文本构成，要分析出的是被空格、tab 和换行符分隔的连续的字符序列。解决此问题的一个简单的方法是：将输入文件分成行，把每一行按顺序分给线程：第一行给线程0，第二行给线程1，…，第 $t$ 行给线程 $t-1$，第 $t+1$ 行给线程0等。

我们可以通过使用信号量将访问输入行串行化。接着，在一个线程读了一行输入后，它就能够对这一行进行分词。一个方法是使用 string.h 中的 strtok 函数，它的原型如下：

```
char* strtok(
      char*       string      /* in/cut */,
      const char* separators  /* in     */);
```

它的用法有些不同寻常：第一次调用它时，string 参数应该是要被分词的文本，在我们的例子
195 中就是一行输入。后面调用它时，string 参数应该为 NULL。它的思想是：在第一次调用时，strtok 缓存一个指向 string 的指针，在接下来连续的调用中，strtok 返回从缓存的副本中分隔出的连续的词。将词分隔开的分隔符通过 separators 参数传入函数。应该将字符串" \t\n" 作为 separators 参数传入函数。

---

⊖ 这部分内容在第5章中也有论述，如果你已经读过那一章，你可以跳过这部分。

**程序 4-14　多线程分词器的第一个版本的程序**

```
1  void* Tokenize(void* rank) {
2     long my_rank = (long) rank;
3     int count;
4     int next = (my_rank + 1) % thread_count;
5     char *fg_rv;
6     char my_line[MAX];
7     char *my_string;
8
9     sem_wait(&sems[my_rank]);
10    fg_rv = fgets(my_line, MAX, stdin);
11    sem_post(&sems[next]);
12    while (fg_rv != NULL) {
13       printf("Thread %ld > my line = %s", my_rank, my_line);
14
15       count = 0;
16       my_string = strtok(my_line, " \t\n");
17       while ( my_string != NULL ) {
18          count++;
19          printf("Thread %ld > string %d = %s\n", my_rank, count,
20                my_string);
21          my_string = strtok(NULL, " \t\n");
22       }
23
24       sem_wait(&sems[my_rank]);
25       fg_rv = fgets(my_line, MAX, stdin);
26       sem_post(&sems[next]);
27    }
28
29    return NULL;
30 }  /* Tokenize */
```

给定以上假设，我们编写如程序 4-14 所示的线程函数。主线程初始化一个有 $t$ 个信号量的数组（每个线程一个）。线程 0 的信号量初始化为 1，所有其他的信号量初始化为 0。第 9 ~ 11 行的代码强制线程按顺序访问输入行。线程首先读入第一行，此时所有其他的线程阻塞在 sem_wait 上。当线程 0 执行 sem_post 时，线程 1 能读一行输入。在每个线程已经读了它的第一行输入后（或已经达到文件尾），更多额外的输入会在第 24 ~ 26 行读入。fgets 函数用于读一行输入，而 [196] 第15 ~ 22 行用于识别这一行的词。当用一个线程运行这个程序时，它能正确地对输入流进行分词。我们第一次用 2 个线程运行这个程序，输入为：

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

输出也是正确的。然而，我们用同样的输入再次运行这个程序，得到下列的输出：

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

这是怎么回事？再回头看一下 strtok 如何对输入行进行缓存。它通过声明一个 storge 类的静态变量来实现对输入行的缓存。导致从本次调用到下次调用之间，存储在这个变量中的值会一直被保留。但不幸的是，这个缓存的字符串是共享的，而不是私有的。因此，线程 0 调用 strtok 对输入的第 3 行进行缓存，覆盖了原来线程 1 调用 strtok 读入输入的第 2 行的缓存。

strtok 函数不是线程安全的：如果多个线程同时调用它，输出可能是不正确的。遗憾的是，对于 C 语言的库函数来说，线程不安全是常见的。例如，stdlib.h 中的随机数生成器 random 和 time.h 中的时间转换函数 localtime 都不是线程安全的。在某些情况下，C 标准库指定一个替代的、线程安全的函数版本。事实上，strtok 有一个线程安全的版本：

```
char* strtok_r(
      char*       string      /* in/out */,
      const char* separators  /* in     */,
      char**      saveptr_p   /* in/out */);
```

"_r" 表示函数是可重入的，"可重入" 有时用做线程安全的同义词。函数的前两个参数与 strtok 是一样的。strtok_r 使用添加_p 的 saveptr 参数跟踪函数在输入字符串中的位置，它起
[197] 到 strtok 函数中缓存指针的作用。通过用 strtok_r 替代 strtok，可以更正原来的 Tokenize 函数。只需要简单地声明一个 char * 变量并传入函数 strtok_r 的第三个参数，分别用下列调用取代原来程序中的第 16 行和第 21 行：

```
my_string = strtok_r(my_line, " \t\n", &saveptr);
. . .
my_string = strtok_r(NULL, " \t\n", &saveptr);
```

### 不正确的程序能产生正确的输出

我们注意到，原来的分词程序的程序错误特别隐匿。第一次用 2 个线程运行程序时，程序产生正确的输出。直到之后的再次运行，我们才发现一个错误。不幸的是，这在并行程序中不是偶然出现的，在共享内存程序中特别常见。因为对大部分程序，线程是互相独立运行的，正如我们前面看到的那样，被执行的语句序列是非确定的。例如，我们不知道线程 1 什么时候第一次调用 strtok。如果它的第一次调用发生在线程 0 已经对第 1 行分词之后，那么第 1 行被标识的词应该是正确的。然而，如果线程 1 在线程 0 完成第 1 行的分词之前调用 strtok，线程 0 就可能不能够完全识别第 1 行所有的词。因此，在开发共享内存程序时，不要因为一个程序产生了正确的输出，就认定它一定是正确。我们总要谨慎地处理竞争条件。

## 4.12 小结

与 MPI 一样，Pthreads 是程序员用来实现并行程序的函数库。与 MPI 不同，Pthreads 用于实现共享内存并行性。

在共享内存编程中，**线程**相当于分布式内存编程中的进程。然而，线程经常比进程更轻量级。

在 Pthreads 程序中，所有的线程都能访问全局变量，但是局部变量对于运行程序的线程来说是私有的。为了使用 Pthreads，我们应该导入 pthread.h 头文件，并且在编译程序时，通过在命令行添加 -lpthread 为程序链接 Pthread 库。我们分别使用函数 pthread_create 和 pthread_join 来启动和停止一个线程。

当多个线程同时执行时，多个线程执行语句的顺序通常是非确定的。当多个线程试图访问一个共享资源时，例如一个共享变量或一个共享文件，并且其中至少有一个访问是更新操作，这样
[198] 的访问可能会导致错误，导致结果的不确定性，我们称这种现象为**竞争条件**（race condition）。编

写共享内存程序时最重要的任务之一就是识别和更正竞争条件。**临界区**（critical section）是一个代码块，在这个代码块中，任意时刻只有一个线程能够更新共享资源，因此临界区中的代码执行应该作为串行代码执行。因此，在设计程序时，应尽可能少地使用临界区，并且使用的临界区应该尽可能短。

有三种避免对临界区竞争访问的基本方法：忙等待、互斥量和信号量。**忙等待**（busy-waiting）可以用一个标志变量和一个空循环来实现。但它十分浪费 CPU 周期。如果打开编译器优化，它又是不可靠的，因此一般来说使用互斥量和信号量会更好。

**互斥量**（mutex）可以被看做是临界区的一把锁，因为互斥量可以保证对临界区的互斥访问。在 Pthreads 中，线程通过调用 `pthread_mutex_lock` 来获取互斥量（锁），并通过调用 `pthread_mutex_unlock` 来释放互斥量（锁）。当一个线程试图获取一个已经在使用的互斥量时，它就阻塞在 `pthread_mutex_lock` 上。这意味着该线程会一直空闲直到系统给它锁。**信号量**（semaphore）是一个有两个操作（`sem_wait` 和 `sem_post`）的无符号型整数。如果信号量是正的，对 `sem_wait` 的调用就简单地将信号量减 1；如果信号量是零，调用 `sem_wait` 的线程就会阻塞直到信号量为正数，此时信号量会减 1，然后线程从调用中返回。`sem_post` 操作使信号量加 1，所以信号量可以当做互斥量使用，其中，`sem_wait` 对应 `pthread_mutex_lock`，`sem_post` 对应 `pthread_mutex_unlock`。然而，信号量比互斥量功能更强，因为它们能够初始化为任何非负值。而且，因为信号量没有“归属权”，任何线程都能够对锁上的信号量进行解锁。信号量能够很容易地用来实现**生产者 - 消费者同步**。在生产者 - 消费者同步中，一个“消费者”线程在继续运行前需要等待一些条件或数据被“生产者”线程创建。信号量不是 Pthreads 的一部分。为了使用它们，需要导入 `semaphore.h` 头文件。

一个**路障**（barrier）是程序中的一个结点，线程必须阻塞直到所有的线程都到达了这个结点。有几种不同的构造路障的方法。其中一种使用了条件变量。**条件变量**（conditional variable）是一个特殊的线程对象，它用来挂起一个线程的执行直到某个条件发生。一旦条件发生，另一个线程能够用一个条件信号或一个条件广播唤醒挂起的线程。

最后介绍的 Pthreads 构造是**读写锁**（read-write lock）。当多个线程同时安全地读一个数据结构时，可以使用读写锁；但如果线程需要修改或者写数据结构时，在修改期间，只有一个线程能够访问该数据结构。

现代微处理器结构使用缓存来减少内存的访问时间，因此典型的处理器结构有特殊的硬件来保证不同芯片上的缓存是**一致的**（coherent）。因为缓存一致的基本单位是一个**缓存行**（cache line）或**缓存块**（cache block），它通常比一个存储字大，所以这可能会带来负效应：两个线程可能正在访问不同的内存单元，当两个单元属于同一个缓存行时，缓存一致性硬件会看成这两个线程正在访问同一个内存单元。这样，如果一个线程更新了它的内存单元，之后其他的线程试图读它想访问的内存单元（与前面那个线程更新的内存单元同在一个缓存行），就不得不从主存中获取值。就是说，缓存一致性硬件强迫多个线程看起来好像是共享同一个内存单元。这称为**伪共享**（false sharing），它会严重地降低共享内存程序的性能。 199

某些 C 语言函数通过声明 `static` 变量从而在两次调用之间缓存数据。当多个线程调用该函数时，可能会引起错误，因为在线程之间共享静态存储，所以一个线程能够覆盖另一个线程的数据。这样的函数不是**线程安全的**（thread-safe）。不幸的是，在 C 语言库中有不少这样的函数。然而，有时会有一些保证线程安全的变形函数可以替用。

观察使用线程不安全函数的程序，我们发现有些问题特别隐匿：使用多个线程和固定输入运行程序时，即使程序是错误的，它有时也会产生正确的输出。这意味着即使在测试中程序产生了正确的输出，也不保证它实际上是正确的，要靠我们自己来识别可能的竞争条件。

## 4.13 习题

4.1 当讨论矩阵 - 向量乘法时，我们通常假设 $m$ 和 $n$，即矩阵的行数和列数，都能够被 $t$ 整除，$t$ 是线程的个数。但是，如果 $m$ 和 $n$ 不满足能被 $t$ 整除的条件，那么用什么公式来分配数据？

4.2 如果想要物理上分割一个数据结构给多个线程中，也就是说，想让数据结构的不同成员对各个线程局部化，我们至少需要考虑三个问题：

a. 数据结构的成员怎样被单个线程独立使用？

b. 数据结构在哪里初始化？怎样初始化？

c. 在数据结构的成员计算后，在哪里使用数据结构？怎样使用？

我们先简单地看一下在矩阵 - 向量乘法函数中的第一个问题。我们看到整个向量 x 被所有的线程使用，很明显它应该被共享。然而，对于矩阵 A 和乘积向量 y，问题（a）似乎是建议将 A 和 y 的元素分布在多个线程中。

为了在线程中划分 A 和 y，我们要做什么？划分 y 不是很难，每个线程可以分配一块内存区来存储分
200 配给它的 y 元素。同样，对矩阵 A，每个线程分配一块内存区来存储分配给它的 A 的部分行。请你修改矩阵 - 向量乘法程序，让程序可以分割这两个数据结构并将它们分配给各个线程。你可以“调度”输入和输出使得线程能读入 A 并且打印出 y 吗？A 和 y 的分配方案是否会影响矩阵 - 向量乘法的运行时间？（运行时间不包括输入和输出）。

4.3 编译器无法知道一个普通的 C 程序是否是多线程，因此，可能会做出干扰忙等待的编译优化。（注意：编译器优化不应该影响互斥量、条件变量或信号量。）与其完全关闭编译器优化，不如使用另一个替代方法：用 C 关键字 volatile 来标识一些共享变量，告诉编译器这些变量可能会被多个线程共享。这样，编译器就知道对涉及这些变量的语句不要应用优化。下面这段代码举例说明了当多个线程试图把一个私有变量与一个共享变量相加求和时，对竞争条件的忙等待解决方案：

```
/* x and flag are shared, y is private          */
/* x and flag are initialized to 0 by main thread */

y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

从这段代码中，编译器看不出 while 语句和 x = x + y 顺序的重要性。因为如果代码是单线程的，这两个语句的顺序不影响代码的结果。但如果编译器决定通过互换这两个语句的顺序来提高寄存器的利用率，结果可能是错误的。

但如果我们把定义：

```
int flag;
int x;
```

替换为定义：

```
int volatile flag;
int volatile x;
```

编译器就知道 x 和 flag 会被其他线程更新，它就不会重排语句的顺序。

对于 gcc 编译器，缺省情况下不做编译优化。你也可以在编译命令行添加 -O0 选项来确保一定不做优化。你可以试试看，运行使用忙等待的无编译优化的 π 计算程序（pth_pi_busy.c），看看多线程计算和单线程计算相比结果如何？再尝试使用编译优化来运行这个程序。在使用 gcc 时，用 -O2 选
201 项取代 -O0 选项。如果发现了程序错误，此时你使用的是几个线程运行该程序？

在 π 计算程序中，哪些变量应该标记为 volatile？改变这些变量的定义，把它们标记为 volatile，并分别使用和不使用优化重新运行该程序，与单线程程序相比，结果如何？

4.4 如果增加线程的个数，直至超过可使用的 CPU 数目，我们发现使用互斥量的 π 计算程序的性能几乎保

持不变。这个现象说明应该如何在可用的处理器上调度线程?

4.5 请修改使用互斥量的 π 计算程序，使临界区在 for 循环内。这个版本的性能与原来的忙等待版本相比如何？我们怎样解释它？

4.6 请修改使用互斥量的 π 计算程序，使得它使用信号量取代互斥量。这个版本的性能与互斥量版本相比如何?

4.7 尽管生产者 - 消费者同步采用信号量很容易实现，但它也能用互斥量来实现。基本的想法是：让生产者线程和消费者线程共享一个互斥量。用一个被主线程初始化为 false 的标志变量来表示是否有产品可以被“消费”。这两个线程的执行如下：

```
while (1) {
   pthread_mutex_lock(&mutex);
   if (my_rank == consumer) {
      if (message_available) {
         print message;
         pthread_mutex_unlock(&mutex);
         break;
      }
   } else { /* my_rank == producer */
      create message;
      message_available = 1;
      pthread_mutex_unlock(&mutex);
      break;
   }
   pthread_mutex_unlock(&mutex);
}
```

如果消费者线程首先进入循环，它会看到没有可用的信息（message_available 值为 false）并在调用 pthread_mutex_unlock 后返回。消费者线程重复上述过程，直到生产者线程生产出信息。请编写一个双线程程序，实现这个版本的生产者 - 消费者同步。你可以将这个程序一般化吗？让它能够运行 $2k$ 个线程，其中奇数线程是消费者，偶数线程是生产者。另外，将程序一般化为每个线程既是生产者又是消费者，你能做到吗？例如，线程 $q$ 既要发送一条信息给线程 $(q+1) \bmod t$，又要从线程 $(q-1+t) \bmod t$ 接收一条信息，如何编写程序？要使用忙等待吗？ 202

4.8 如果一个程序使用超过一个互斥量，并能够以不同的顺序来获取互斥量，程序可能会死锁。也就是说，线程可能会永远地阻塞等待获取一个锁。例如，假设一个程序有两个共享数据结构（如，两个数组或者两个链表），每个数据结构有一个与其相关联的互斥量（锁）。而且，我们假设每个数据结构能够在线程获取数据结构关联的互斥量后被访问（读或修改）。

a. 用两个线程运行程序。假设发生了下列顺序的事件：

| 时间 | 线程 0 | 线程 1 |
|---|---|---|
| 0 | pthread mutex lock (&mut0) | pthread mutex lock (&mut1) |
| 1 | pthread mutex lock (&mut1) | pthread mutex lock (&mut0) |

会发生什么?

b. 如果程序使用忙等待（采用两个标志变量）替代互斥量，会有问题吗?

c. 如果程序使用信号量替代互斥量，会有问题吗?

4.9 以下是一些 Pthreads 定义的路障函数的实现。函数：

```
int pthread_barrier_init(
      pthread_barrier_t*            barrier_p  /* out */,
      const pthread_barrierattr_t*  attr_p     /* in  */,
      unsigned                      count      /* in  */);
```

用于初始化一个路障对象 barrier_p。一般情况下，我们会忽略第二个参数，只传递 NULL 值给这个参数。最后一个参数指定在线程能够继续运行前必须到达路障的线程数。路障本身是对以下函数的

调用：

```
int pthread_barrier_wait(
      pthread_barrier_t*  barrier_p  /* in/out */);
```

与大部分其他的 Pthreads 对象一样，有一个释放函数：

```
int pthread_barrier_destroy(
      pthread_barrier_t*  barrier_p  /* in/out */);
```

修改本书网站上的一个路障程序，让它使用 Pthreads 路障。找一个支持 Pthreads 路障实现的系统，并用不同数量的线程在系统上运行你的程序。这个程序的性能与其他实现相比如何？

4.10 修改你在之后的编程作业中编写的一个程序，让它使用 4.8 节的机制来对自己计时。为了计算经过
203 的时间，你可以使用本书网站上介绍的在头文件 timer.h 中的宏 GET_TIME。注意这个宏将给出墙上时钟（wall clock）时间，不是 CPU 时间。还要注意：因为它是宏，所以它能直接操作它的参数。例如，为了实现：

```
Store current time in my_start;
```

你可以使用：

```
GET_TIME(my_start);
```

而不是：

```
GET_TIME(&my_start);
```

你将怎样实现路障？你又将怎样实现下列伪代码？

```
elapsed = Maximum of my_elapsed values;
```

4.11 考虑一个链表以及对链表进行的访问操作，下列的哪些操作可能会导致问题：

a. 两个 Delete 操作同时执行。

b. 一个 Insert 和一个 Delete 操作同时执行。

c. 一个 Member 和一个 Delete 操作同时执行。

d. 两个 Insert 同时执行。

e. 一个 Insert 和一个 Member 同时执行。

4.12 链表操作 Insert 和 Delete 都由两个不同的阶段组成。在第一阶段，这两个操作要么查找新结点的位置，要么查找要删除结点的位置。在第一阶段的输出结果确定后，在第二阶段要么插入一个新结点，要么删除一个存在的结点。其实，对链表程序来说，把这种类型的操作分成两个函数调用是十分常见的。对于这两个操作，第一阶段都只涉及对链表的读访问，只有第二个阶段才修改链表。如果在第一阶段使用一个读锁来锁链表是否安全？在第二阶段用写锁来锁链表是否安全？请解释你的答案。

4.13 请从网站上下载多线程的链表程序。在我们的例子中，查找操作所占的比例是固定的，剩下的部分比例由插入与删除操作构成。

a. 重新运行实验，实验包括所有的查找和插入操作。

b. 重新运行实验，实验包括所有的查找和删除操作。

两次实验总的运行时间上有什么不同？插入和删除操作哪一个更耗时？

4.14 在 C 语言中，将二维数组作为参数的函数必须在参数列表中指定数组的列数。因此对于 C 程序员来说，只使用一维数组是很常见的，通常编写一段显式的代码将二维数组一对下标转换为对应一维数
204 组的下标。修改 Pthreads 的矩阵 - 向量乘法程序，用一个一维数组表来示矩阵，并调用矩阵 - 向量乘法函数。这样的改变会影响运行时间吗？

4.15 请从本书的网站上下载源文件 pth_vect_rand_split.c。找一个可以对缓存进行概要分析的程序（例如 Valgrind 的程序 [49]），并根据缓存概要分析文档的指令来编译程序（需要符号表和完全编译优化 gcc -g -O2...）。现在根据文档的指令运行程序，使用三种不同 的输入 $k \times (k \cdot 10^6)$、$(k \cdot$

$10^3$）×（$k \cdot 10^3$）和（$k \cdot 10^6$）×$k$。选择较大的 $k$，使这三个输入中至少有一个的二级缓存缺失数达到 $10^6$ 的数量级。

a. 这三种输入所引起的一级缓存写缺失各为多少？
b. 这三种输入所引起的二级缓存写缺失各为多少？
c. 部分的写缺失发生在哪里？哪种输入的写缺失最多？你能解释为什么吗？
d. 这三种输入所引起的一级缓存读缺失各为多少？
e. 这三种输入所引起的二级缓存读缺失各为多少？
f. 大部分的读缺失发生在哪里？哪种输入的读缺失最多？你能解释为什么吗？
g. 分别采用这三种输入运行程序，但不使用缓存分析器。在哪个输入下程序运行最快？哪个输入下程序运行最慢？你对缓存失效的观察能解释这个差别吗？如何解释？

4.16 在矩阵 - 向量乘法的例子中，采用 8000 × 8000 的输入。假设程序用 4 个线程运行，线程 0 和线程 2 被分配到不同的处理器上运行。如果一个缓存行大小为 64 字节或 8 个 double 型数，在线程 0 和线程 2 之间会对向量 y 的任何一部分发生伪共享吗？为什么？如果线程 0 和线程 3 被分配到不同的处理器会怎样？它们之间会对 y 的任何一部分发生伪共享吗？

4.17 在矩阵 - 向量乘法的例子中，采用 8 × 8 000 000 的输入。假设 double 型数据占用 8 字节的存储空间，一个缓存行大小为 64 字节。同时假设系统有 2 个双核处理器。

a. 最少需要多少个缓存行来存储向量 y？
b. 最多需要多少个缓存行来存储向量 y？
c. 如果缓存行的边界总是按照 8 字节的 double 型对齐，有多少种不同的方式给 y 的元素分配缓存行？
d. 如果我们只考虑一对线程共享一个处理器，可以有多少种不同的方式将 4 个线程分配到处理器上？ 205
这里我们假设同一个处理器上的所有核共享一个缓存。
e. 在我们的例子中，是否有一个向量元素到缓存行的分配，以及线程到处理器的分配方式，使得没有伪共享发生吗？换句话说，是否能做到分配到一个处理器的多个线程，它们各自分配到的 y 的元素能存储在同一个缓存行中，在其他处理器上运行线程所分配到的 y 的元素存储在不同的缓存行上？
F. 有多少种向量元素到缓存行、线程到处理器的分配方式？
g. 在这些分配方式中，有多少种不会导致伪共享发生？

4.18 a. 修改矩阵 - 向量乘法程序，在可能发生伪共享时填充向量 y。填充 y 的目的是，如果线程按锁步执行，包含 y 的元素的缓存行不可能被两个或更多个线程共享。例如，假设一个缓存行可以存储 8 个 double 型数，我们用 4 个线程执行程序。如果我们为 y 分配至少 48 个 double 型数的存储空间，那么，在每次 for i 循环时不可能有 2 个线程同时访问同一个缓存行。
b. 修改矩阵 - 向量乘法程序，让每个线程在 for i 循环中使用私有空间来存储它使用的 y 的那部分。当一个线程计算完它自己的那部分 y 后，再把它们从私有空间复制到到共享变量中去。
c. 与原始程序相比，这两个替代方案的性能如何？它们之间相比又如何？

4.19 尽管 strtok_r 是线程安全的，但它有一个不太好的特性，就是它会无端地修改了输入的字符串。请编写一个线程安全并且不修改输入字符串的分词程序。

## 4.14 编程作业

4.1 编写一个 Pthreads 程序，实现第 2 章的直方图程序。

4.2 假设我们向一个正方形的标靶上随机投掷飞镖，靶心在正中央，标靶的长和宽都是 2 英尺。同时假设有一个圆与标靶内切。圆的半径是 1 英尺，面积是 π 平方英尺。如果击中点在标靶上是均匀分布的（我们总会击中正方形），那么飞镖击中圆的数量近似满足等式

$$\frac{\text{圆中击中点的数量}}{\text{总的投掷数}} = \frac{\pi}{4}$$

206 因为圆的面积与正方形的面积之比为 $\pi/4$。

我们可以使用这个公式和一个随机数生成器来估算 $\pi$ 的值。

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

这称为“蒙特卡洛”方法，因为它利用了随机性（投飞镖）。

编写一个 Pthreads 程序，使用蒙特卡洛方法估算 $\pi$。由主线程读入总的投掷数，然后输出估算值。可能需要对圆的命中次数和投掷次数都使用 `long long int`（长整）型，为了对 $\pi$ 值进行合理的估算，这两个值必须足够大。

4.3 编写一个 Pthreads 程序实现梯形积分。使用一个共享变量来表示所有线程计算结果的总和。如果使用忙等待、互斥量和信号量来保证临界区的互斥。每个方法的优点和缺点分别是什么？

4.4 编写一个 Pthreads 程序，计算你的系统创建和终止一个线程所需要的平均时间。线程的数量会影响平均时间吗？如果是，为什么？

4.5 编写一个 Pthreads 程序实现一个“任务队列”。主线程启动用户指定数量的线程，这些线程会在因为等待某个条件而立即睡眠。主线程还生成由其他线程执行的任务块；每次它生成一个新的任务块，就会用一个条件信号唤醒一个线程。当一个线程完成任务块的执行时，它又会回到条件等待。当主线程完成了所有的生成任务后，它会设置某个全局变量，指示再也没有更多的任务生成了，并用一个条件广播唤醒所有线程。为了清晰起见，将任务采用链表操作。

4.6 编写一个 Pthreads 程序，使用两个条件变量和一个互斥量来实现一个读写锁。下载使用 Pthreads 读写锁的在线链表程序，修改该程序，让它使用你的读写锁。比较当读优先级更高时和写优先级更高时程
207 序的性能。并进行归纳总结。

# 用 OpenMP 进行共享内存编程

与 Pthreads 一样，OpenMP 是一个针对共享内存并行编程的 API。OpenMP 中的“MP”代表“多处理”，是一个与共享内存并行编程同义的术语。因此，OpenMP 是为此类系统而设计的：在系统中每个线程或进程都有可能访问所有可访问的内存区域。当使用 OpenMP 编程时，我们将系统看做一组核或 CPU 的集合，它们都能访问主存，如图 5-1 所示。

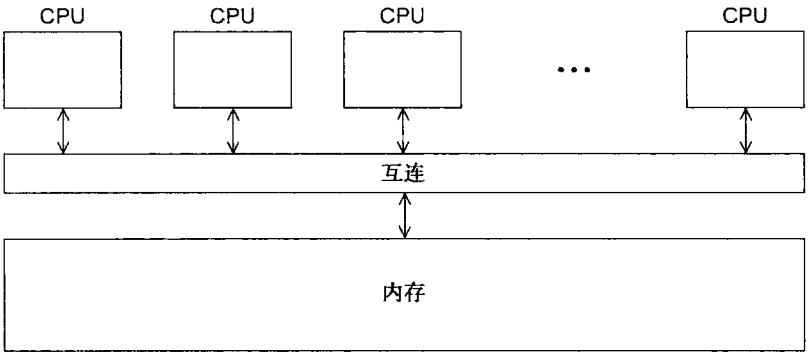



图 5-1　一个共享内存的系统

尽管 OpenMP 和 Pthreads 都是针对共享内存编程的 API，但它们有许多本质的不同。Pthreads 要求程序员显式地明确每个线程的行为。相反，OpenMP 有时允许程序员只需要简单地声明一块代码应该并行执行，而由编译器和运行时系统来决定哪个线程具体执行哪个任务。这也意味着 OpenMP 和 Pthreads 还有另一个不同之处，即 Pthreads（与 MPI 一样）是一个能够被链接到 C 程序的函数库，因此只要系统有 Pthreads 库，Pthreads 程序就能够被任意 C 编译器使用。相反，OpenMP 要求编译器支持某些操作，所以完全有可能你使用的编译器无法把 OpenMP 程序编译成并行程序。

这些不同也说明了为什么共享内存编程会有两个标准 API：Pthreads 更底层，并且提供了虚拟地编写任何可知线程行为的能力。然而，这个功能有一定的代价：每个线程行为的每一个细节都得由我们自己来定义。相反，OpenMP 允许编译器和运行时系统来决定线程行为的一些细节，因此使用 OpenMP 来编写一些并行行为更容易。但代价是很难对一些底层的线程交互进行编程。

程序员和计算机科学家开发 OpenMP 的原因是：他们认为使用诸如 Pthreads 的 API 来编写大规模高性能的程序实在太难了。他们定义了 OpenMP 规范，在一个更高的层次上开发共享内存程序。事实上，OpenMP 明确地被设计成可以用来对已有的串行程序进行增量式并行化，这对于 209
MPI 是不可能的，对于 Pthreads 也是相当困难的。

本章我们将学习 OpenMP 的基础知识，学习如何使用 OpenMP 编写程序，以及如何编译和运行 OpenMP 程序。接下来，我们还会学习怎样利用 OpenMP 最强大的功能中的一个：只需要对源代码进行少量改动就可以并行化许多串行的 `for` 循环。我们还会看到 OpenMP 的一些其他特征：任务并行化和显式线程同步。我们也会看到一些在共享内存编程中的标准问题：缓存对共享内存编程的影响，以及当串行代码（特别是一个串行库）被一个共享内存程序使用时遇到的问题。

## 5.1 预备知识

OpenMP 提供“基于指令”的共享内存 API。这意味着：在 C 和C ++ 中，有一些特殊的预处理器指令 pragma。在系统中加入预处理器指令一般是用来允许不是基本 C 语言规范部分的行为。不支持 pragma 的编译器就会忽略 pragma 指令提示的那些语句，这样就允许使用 pragma 的程序在不支持它们的平台上运行。因此，在理论上，如果你仔细编写一个 OpenMP 程序，它就能够在任何有 C 编译器的系统上被编译和运行，而无论编译器是否支持 OpenMP。

在 C 和C ++ 中，预处理器指令以#pragma 开头。通常，我们把字符#放在第一列，并且像其
210 他预处理器指令一样，移动指令的剩余部分，使它和剩下的代码对齐。与所有的预处理器指令一样，pragma 的默认长度是一行，因此如果有一个 pragma 在一行中放不下，那么新行需要被“转义”——前面加一个反斜杠“\”。#pragma 后面要跟什么内容，完全取决于正在使用哪些扩展。

看一个十分简单的例子——一个使用 OpenMP 的“hello，world”程序，如程序 5-1 所示。

**程序 5-1 一个使用 OpenMP 的“hello，world”程序**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Hello(void);  /* Thread function */
6
7  int main(int argc, char* argv[]) {
8     /* Get number of threads from command line */
9     int thread_count = strtol(argv[1], NULL, 10);
10
11 #  pragma omp parallel num_threads(thread_count)
12    Hello();
13
14    return 0;
15 }  /* main */
16
17 void Hello(void) {
18    int my_rank = omp_get_thread_num();
19    int thread_count = omp_get_num_threads();
20
21    printf("Hello from thread %d of %d\n", my_rank, thread_count);
22
23 }  /* Hello */
```

### 5.1.1 编译和运行 OpenMP 程序

为了用 gcc 编译这个程序，需要包含 - fopenmp 选项[⊖]：

```
$ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

为了运行程序，在命令行中明确线程的个数。例如，希望有四个线程运行程序，就输入：

211 `$ ./omp_hello 4`

如果这样做，输出可能是：

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

⊖ 有些老版本的 gcc 可能不包含 OpenMP 支持。一般而言，其他的编译器使用不同的命令行选项来明确源程序是否是一个 OpenMP 程序。要获取关于编译器使用的细节，请看 2.9 节。

然而，应该注意到线程正在竞争访问标准输出，因此不保证输出会按线程编号的顺序出现。例如，输出也可能是：

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

或

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

或者任何其他的线程编号的排列。

如果我们只想用一个线程运行程序，可以输入：

```
$ ./omp_hello 1
```

我们将得到输出：

```
Hello from thread 0 of 1
```

### 5.1.2　程序

我们来看一下程序 5-1 中的源代码。除了指令集合外，OpenMP 由一个函数和宏库组成，因此我们通常需要包含一个有原型和宏定义的头文件。OpenMP 的头文件是 omp.h，程序的第 3 行包含了它。

在 Pthreads 程序中，我们在命令行里指定线程数，OpenMP 程序也经常这么做。因此在第 9 行，使用 stdlib.h 中的 strtol 函数来获得线程数。这个函数的语法是：

```
long strtol(
      const char*  number p     /* in  */,
      char**       end p        /* out */,
      int          base         /* in  */);
```

第一个参数是一个字符串（在我们的例子中，它是命令行参数），最后的参数是字符串所表示的数的基数（在我们的例子中是 10（十进制））。我们不使用第二个参数，因此只是传入一个 NULL（空）指针。

212

如果你已经了解 C 语言编程，那么到这里为止我们都没有介绍新的内容。当我们从命令行启动程序时，操作系统启动一个单线程的进程，进程执行 main 函数中的代码。然而，在程序的第 11 行，事情变得有趣了。这是第一条 OpenMP 指令，使用它来提示程序应该启用一些线程。每个被启动的线程都执行 Hello 函数，并且当线程从 Hello 调用返回时，它们应该被终止，即在执行 **return** 语句时进程应该被终止。

程序的代码上有许多重大的改变。如果你学习了第 4 章，就会想起我们必须写许多代码来派生（fork）和合并（join）多个线程：需要为每个线程的特殊结构分配存储空间，需要使用一个 **for** 循环来启动每个线程，并使用另一个 **for** 循环来终止这些线程。因此，很明显 OpenMP 比 Pthreads 层次更高。

我们已经看到：在 C 和C ++ 中，pragma 总以

```
## pragma
```

作为开始。

OpenMP 的 pragma 总是以

```
## pragma omp
```

作为开始。

在 pragma 后面的第一条指令是一条 parallel 指令，你也许已经猜到：使用 parallel 是用来表明之后的**结构化代码块**（structured block，也可以称为**基本块**）应该被多个线程并行执行。一个结构化代码块是一条 C 语句或者只有一个入口和一个出口的一组复合 C 语句，但在这个代码块中允许调用 exit 函数。这个定义简单地禁止分支语句进入或离开结构化代码块。

**线程**（thread）是执行线程（thread of execution）的简写。这个名字表示被一个程序执行的一系列语句。典型地，线程被同一个进程**派生**（fork），这些线程共享启动它们的进程的大部分资源（例如，对标准输入和标准输出的访问），但每个线程有它自己的栈和程序计数器。当一个线程完成了执行，它就又**合并**（join）到启动它的进程中。线程在示意图中表示为有向线段。如图 5-2 所示。更多的细节见第 2 章和第 4 章。

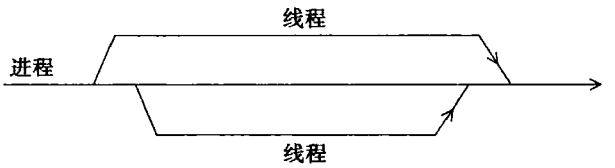


图 5-2 一个派生和合并两个线程的进程

最基本的 parallel 指令可以以如下简单的形式表示：

213
```
#  pragma omp parallel
```

运行结构化代码块的线程数将由运行时系统决定。这里使用的算法十分复杂，细节见 OpenMP 标准 [42]。如果没有其他线程启动，典型情况下系统将在每个核上运行一个线程。

如前面提到的那样，通常会在命令行里指定线程数，因此为 parallel 指令增加 num_threads 子句。在 OpenMP 中，**子句**只是一些用来修改指令的文本。num_threads 子句被添加到 parallel 指令中，这样就允许程序员指定执行后代码块的线程数：

```
#  pragma omp parallel num_threads(thread_count)
```

需要注意的是，程序可以启动的线程数可能会受系统定义的限制。OpenMP 标准并不保证实际情况下能够启动 thread_count 个线程。然而，目前大部分的系统能够启动数百甚至数千个线程，因此除非你试图启动许多线程，否则一般情况下我们几乎总能够得到需要数目的线程。

当程序到达 parallel 指令时，究竟会发生什么？在 parallel 之前，程序只使用一个线程。而当程序开始执行时，进程开始启动。当程序到达 parallel 指令时，原来的线程继续执行，另外 thread_count −1 个线程被启动。在 OpenMP 语法中，执行并行块的线程集合（原始的线程和新的线程）称为线程**组**（team），原始的线程称为**主线程**（master），额外的线程称为**从线程**（slave）。每个线程组中的线程都执行 parallel 指令后的代码块，因此在我们的例子中，每个线程都调用 Hello 函数。

当代码块执行完时，即在我们的例子中，当线程从 Hello 调用中返回时，有一个**隐式路障**。这意味着完成代码块的线程将等待线程组中的所有其他线程完成代码块——在我们的例子中，一个已经完成 Hello 调用的线程将等待线程组中所有其他线程返回。当所有线程都完成了代码块，从线程将终止，主线程将继续执行之后的代码。在我们的例子中，主线程将执行第 14 行的 return语句，程序将终止。

因为每个线程有它自己的栈，所以一个执行 Hello 函数的线程将在函数中创建它自己的私有局部变量。在我们的例子中，当函数被调用时，通过调用 OpenMP 函数 omp_get_thread_num 和 omp_get_num_threads，每个线程将分别得到它的编号或 id，以及线程组中的线程数。线程的编号是一个整数，范围是 0、1、…、thread_count −1。这些函数的语法是：

214
```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
```

因为标准输出被所有线程共享，所以每个线程都能够执行 printf 语句，打印它的线程编号和线程数。与我们早先提到的一样，对于标准输出的访问没有调度，因此线程打印它们结果的实际顺序是不确定的。

### 5.1.3　错误检查

为了使代码更为紧凑、可读性更强，我们的程序不做任何错误检查。当然，这是危险的，而且实际上，试图预测错误并对它们进行检查是一个十分好的主意，甚至可以说是必需的。在这个例子中，首先一定要检查命令行参数的存在，如果存在的话，在调用 strtol 后应该检查值是否是正的。还要检查被 parallel 指令实际创建的线程数与 thread_count 是否一样，但在这个例子中，这并不重要。

第二个潜在问题的来源是编译器。如果编译器不支持 OpenMP，那么它将只忽略 parallel 指令。然而，试图包含 omp.h 头文件以及调用 omp_get_thread_num 和 omp_get_num_threads 将引起错误。为了处理这些问题，可以检查预处理器宏_OPENMP 是否定义。如果定义了，则我们能够包含 omp.h 并调用 OpenMP 函数。我们可能对程序做下列修改。

不只是简单地包含 omp.h

```
#include <omp.h>
```

我们能够在试图包含 omp.h 之前先检查_OPENMP 的定义：

```
#ifdef _OPENMP
#  include <omp.h>
#endif
```

还有，我们可以首先检查是否_OPENMP 定义，从而取代只调用 OpenMP 函数：

```
#  ifdef _OPENMP
      int my_rank = omp_get_thread_num();
      int thread_count = omp_get_num_threads();
#  else
      int my_rank = 0;
      int thread_count = 1;
#  endif
```

这里，如果 OpenMP 无法使用，则 Hello 函数将是单线程的。因此，单线程的编号将是0，线程数将是1。

215

在本书出版社的网站上，有做出这些检查的该版本程序的源代码。为了使程序尽可能清晰，通常会在演示的代码中显示少量的错误检查。

## 5.2　梯形积分法

我们来看一个更实用（也更复杂）的例子：用梯形积分法估计曲线下方所包围的面积。回忆一下在3.2节，如果 $y=f(x)$ 是一个合理的函数，$a<b$ 且都是实数，那么我们能够估计 $f(x)$ 的图形与垂直线 $x=a$、$x=b$ 和 $x$ 轴所围成的区域的面积，方法是将区间 $[a, b]$ 分成 $n$ 个子区间并在每个子区间上使用一个梯形的面积来近似估计区域的面积。见图5-3的例子。

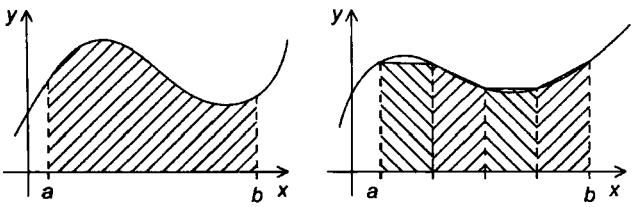
再回忆一下，如果每个子区间有同样的宽度 $h$，并且定义 $h=(b-a)/n$，$x_i=a+ih$，$i=0, 1, \cdots, n$，那么近似值将是：

$$h[f(x_0)/2+f(x_1)+f(x_2)+\cdots+f(x_{n-1})+f(x_n)/2]$$

因此，可以使用下列代码实现梯形积分的串行算法：

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++){
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```

详见 3.2.1 节。




a）　　b）

图 5-3　梯形积分法：a）需要估计的区域；b）使用梯形计算的近似面积

### 第一个 OpenMP 版本

回忆一下，我们应用 Foster 的并行程序设计方法对梯形积分法进行并行化，具体步骤（见 216 3.2.2 节）如下。

（1）识别两类任务：

a. 单个梯形面积的计算。

b. 梯形面积的求和。

（2）在 1（a）的任务中，没有任务间的通信，但这一组任务中的每一个任务都与 1（b）中的任务通信。

（3）假设梯形的数量远大于核的数量，于是通过给每个线程分配连续的梯形块（和每个核一个线程）[⊖]来聚集任务。这能有效地将区间 $[a, b]$ 划分成更大的子区间，每个线程对它的子区间简单地应用串行梯形积分法。例子见图 5-4。

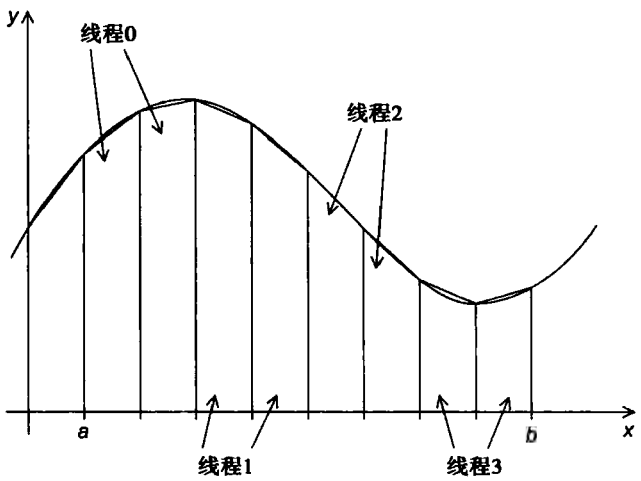



图 5-4　将梯形分配给各个线程

⊖ 我们讨论的这个方法是用在 MPI 程序中的，事实上我们使用的是进程而不是线程。

然而，工作还没有做完，因为还需要累加线程的结果。很明显，其中一个解决方案是使用一个共享变量作为所有线程的和，每个线程可以将它计算的部分结果累加到共享变量中。让每个线程执行类似下面的语句：

```
global_result += my_result;
```

然而，正如我们已经看到的，这可能会导致一个错误的 global_result 值——如果两个（或更多）线程试图同时执行这条语句，那么结果将是不可预计的。例如，假设 global_result 已经被初始化为 0，线程 0 已经计算出 my_result = 1，线程 1 已经计算出 my_result = 2。而且，217
假设线程根据以下时间表执行语句 global_ result + = my_ result：

| 时间 | 线程 0 | 线程 1 |
|---|---|---|
| 0 | global result = 0 送入寄存器 | 完成 my_result |
| 1 | my_result = 1 送入寄存器 | global_result = 0 送入寄存器 |
| 2 | 将 my_result 加到 global result 中 | my_result = 2 送入寄存器 |
| 3 | 存储 global_result = 1 | 将 my_result 加到 global result |
| 4 | | 存储 global_result = 2 |

我们看到线程 0 计算出的值（my_result = 1）被线程 1 覆盖了。

当然，实际运行时，事件的序列可能会不同，但除非一个线程在其他线程开始时完成了计算 global_result + = my_result，否则结果都将是不正确的。这其实是一个**竞争条件**（race condition）的例子：多个线程试图访问一个共享资源，并且至少其中一个访问是更新该共享资源，这可能会导致错误。引起竞争条件的代码 global_result + = my_result，称为**临界区**。临界区是一个被多个更新共享资源的线程执行的代码，并且共享资源一次只能被一个线程更新。

因此需要一些机制来确保一次只有一个线程执行 global_result + = my_result，并且第一个线程完成操作前，没有其他的线程能开始执行这段代码。在 Pthreads 中，使用互斥量或信号量。在 OpenMP 中，使用 critical 指令：

```
#  pragma omp critical
   global_result += my_result;
```

这条指令告诉编译器需要安排线程对下列的代码块进行**互斥**访问，即一次只有一个线程能够执行下面的结构化代码。这个版本的代码在程序 5-2 中。我们已经忽略了任何错误检查，也忽略了函数 $f(x)$ 的代码。

**程序 5-2　第一个 OpenMP 梯形积分法程序**

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <omp.h>
 4
 5  void Trap(double a, double b, int n, double* global_result_p);
 6
 7  int main(int argc, char* argv[]) {
 8     double  global_result = 0.0;
 9     double  a, b;
10     int     n;
11     int     thread_count;
12
13     thread_count = strtol(argv[1], NULL, 10);
14     printf("Enter a, b, and n\n");
15     scanf("%lf %lf %d", &a, &b, &n);
16  #  pragma omp parallel num_threads(thread_count)
```

```
17     Trap(a, b, n, &global_result);
18
19     printf("With n = %d trapezoids, our estimate\n", n);
20     printf("of the integral from %f to %f = %.14e\n",
21        a, b, global_result);
22     return 0;
23  }  /* main */
24
25  void Trap(double a, double b, int n, double* global_result_p) {
26     double  h, x, my_result;
27     double  local_a, local_b;
28     int  i, local_n;
29     int my_rank = omp_get_thread_num();
30     int thread_count = omp_get_num_threads();
31
32     h = (b-a)/n;
33     local_n = n/thread_count;
34     local_a = a + my_rank*local_n*h;
35     local_b = local_a + local_n*n;
36     my_result = (f(local_a) + f(local_b))/2.0;
37     for (i = 1; i <= local_n-1; i++) {
38       x = local_a + i*h;
39       my_result += f(x);
40     }
41     my_result = my_result*h;
42
43  #  pragma omp critical
44     *global_result_p += my_result;
45  }  /* Trap */
```

在 main 函数中，第 16 行之前的代码是单线程的，它简单地获取线程数和输入（$a$、$b$ 和 $n$）。第 16 行里，parallel 指令明确 Trap 函数应该被 thread_count 个线程执行。在从 Trap 调用返回后，任何被 parallel 指令启动的新线程将终止，程序只用一个线程恢复执行。这个线程打印结果并终止。

在 Trap 函数中，每个线程获取它的编号，以及在线程组中被 parallel 指令启动的线程总数。然后，每个线程确定下列值：

(1) 梯形底的长度（第 32 行）。

218 ⋮ 219 (2) 给每个线程分配的梯形数（第 33 行）。

(3) 区间的左、右端点（第 34 行和第 35 行）。

(4) 对 global_result 贡献的部分和（第 36 ~ 41 行）。

在第 43 第和第 44 行，线程通过将它们的部分和结果增加到 global_result 来完成操作。

对某些变量使用前缀 local_来强调它们的值与 main 函数中的对应值不同——例如 local_a 可能与 a 不同，尽管它是线程的左端点。

注意：除非 $n$ 被 thread_count 整除，否则我们将使用小于 $n$ 个的梯形来估算 global_result。例如，如果 $n=14$，thread_count = 4，则每个线程将计算

local_n = n / thread_count = 14/4 = 3

因此每个线程将只使用 3 个梯形，global_result 将由 $4\times3=12$ 个梯形计算出，而不是 14 个。所以在错误检查（在程序中没有显示）时，我们用如下操作来检查 $n$ 是否被 thread_count 整除：

```
if (n % thread_count != 0) {
   fprintf(stderr, "n must be evenly divisible by thread_count\n");
   exit(0);
}
```

因为每个线程分配到了 local_n 个梯形的块，所以每个线程的区间长度将是 local_n * h，故左端点将是：

```
thread 0:  a + 0*local_n*h
thread 1:  a + 1*local_n*h
thread 2:  a + 2*local_h*h
   . . .
```

在第 34 行，进行如下赋值：

```
local_a = a + my_rank*local_n*h;
```

而且，因为每个线程区间的宽度是 local - n * h，所以它的右端点将是

```
local_b = local_a + local_n*h;
```

## 5.3 变量的作用域

在串行编程中，变量的作用域由程序中的变量可以被使用的那些部分组成。例如，在 C 函数开始处被声明的变量有“函数范围”的作用域，即它只能够在函数体中被访问。另一方面，一个 220
在 .c 文件开始处被声明的变量有“文件范围”的作用域，表示任何在文件中声明该变量的函数都能够访问这个变量。在 OpenMP 中，变量的**作用域**涉及在 parallel 块中能够访问该变量的线程集合。一个能够被线程组中的所有线程访问的变量拥有**共享**作用域，而一个只能被单个线程访问的变量拥有**私有**作用域。

在“hello，world”程序中，被每个线程使用的变量（my_rank 和 thread_count）在 Hello 函数中被声明，这个函数在 parallel 块中被调用。结果，被每个线程使用的变量在线程的（私有）栈中分配，因此所有的变量都有私有作用域。这与梯形积分法程序的情况几乎一样，因为 parallel 块只是一个函数调用，因此在 Trap 函数中被每个线程使用的变量在线程的栈中分配。

但是，在 main 函数中声明的变量（a、b、n、global_result 和 thread_count）对于所有线程组中被 parallel 指令启动的线程都是可访问的。因此，在 parallel 块之前被声明的变量的缺省作用域是共享的。事实上，我们已经隐式地使用了每个在线程组中的线程可以从对 Trap 的调用中得到 a、b 和 n 的值。因为这个调用发生在 parallel 块里，所以当它们的值被复制给对应的形式参数时，每个线程都能访问 a、b 和 n，这一点非常重要。

而且，在 Trap 函数中，尽管 global_result_p 是私有变量，但它引用了 global_result 变量，这个变量是在 main 函数中并且在 parallel 指令之前声明的变量。global_result 的值用于存储在 parallel 块之后的打印结果。因此在代码

```
*global_result_p += my_result;
```

中，对于 * global_result_p，拥有共享作用域是很重要的。如果它对每个线程都是私有的，就不需要使用 critical 指令。此外，如果它是私有的，在 parallel 块完成后，将很难在 main 中确定 global_result 的值。

总之，在 parallel 指令前已经被声明的变量，拥有在线程组中所有线程间的共享作用域，而在块中声明的变量（例如，函数中的局部变量）中有私有作用域。另外，在 parallel 块开始处的共享变量的值，与该变量在 parallel 块之前的值一样。在 parallel 块完成后，该变量的值是块结束时的值。

我们将马上看到一个变量的缺省作用域能够用其他指令改变，OpenMP 提供了改变缺省作用域的子句。

## 5.4 归约子句

如果开发实现梯形积分法的串行程序，我们可能会使用一个有些许不同的函数原型，而

不是：

221
```
void Trap(double a, double b, int n, double* global_result_p);
```

我们可能会定义：

```
double Trap(double a, double b, int n);
```

对函数的调用将会是：

```
global_result = Trap(a, b, n);
```

这更容易理解，对于除了指针的忠实拥护者之外的人，这种方法可能更有吸引力。

保留指针版本是因为我们需要将每个线程的局部计算结果加到 global_result。然而，我们可能更倾向于以下的函数原型：

```
double Local_trap(double a, double b, int n);
```

除了没有临界区外，Local_trap 的函数体与程序 5-2 中 Trap 的函数体是一样的。而且，每个线程将返回它这部分的计算，即 my_result 的最终值。如果做出这个改变，可能会改变 parallel 块使之看上去如下所示：

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
#     pragma omp critical
      global_result += Local_trap(double a, double b, int n);
   }
```

你能看出上面这段代码存在的一个问题吗？我们期望它应该能给出正确的答案。但是，因为指定的临界区是：

```
global_result += Local_trap(double a, double b, int n);
```

对 Local_trap 的调用一次只能够被一个线程执行，所以这就相当于强制各个线程顺序执行梯形积分法。如果我们检查这个版本的运行时间，就会发现多个线程运行该程序可能会比一个线程慢（见习题 5.3）。

可以通过在 parallel 块中声明一个私有变量和将临界区移到函数调用之后来避免这个问题：

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */
      my_result += Local_trap(double a, double b, int n);
#     pragma omp critical
      global_result += my_result;
   }
```

现在，对 Local_trap 的调用在临界区之外，这样各个线程能够同时执行对 Local_trap 的调
222 用。而且，因为 my_result 被声明在 parallel 块里，所以它是私有的，在临界区之前每个线程会在它的 my_result 变量中存储它那部分的计算结果。

OpenMP 提供了一个更为清晰的方法来避免 Local_trap 的串行执行：将 global_result 定义为一个*归约*（reduction）变量。**归约操作符**（reduction operator）是一个二元操作（例如：加法和减法），**归约**就是将相同的归约操作符重复地应用到操作数序列来得到一个结果的计算。另外，所有操作的中间结果存储在同一个变量里：**归约变量**（reduction variable）。例如，如果 A 是一个有 n 个 **int** 型整数的数组，计算：

```
int sum = 0;
for (i = 0; i < n; i++)
   sum += A[i];
```

是一个归约，归约操作符是加法。

在 OpenMP 中，可以指定一个归约变量来表示归约的结果。为了能够如此操作，要在 parallel 指令中添加一个 reduction 子句。在我们的例子里，修改代码如下所示。

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

首先，注意 parallel 指令有两行。在 C 语言中，预处理器指令缺省情况下只有一行，因此我们需要通过添加一个反斜杠（\）来转义换行符。

代码明确了 global_result 是一个归约变量，加号（“+”）指示归约操作符是加法。OpenMP 为每个线程有效地创建了一个私有变量，运行时系统在这个私有变量中存储每个线程的结果。OpenMP 也创建了一个临界区，并且在这个临界区中，将存储在私有变量中的值进行相加。因此，对 Local_trap 的调用能够并行执行。

reduction 子句的语法是：

```
reduction(<operator>: <variable list>)
```

在 C 语言中，operator 可能是操作符 +、*、-、&、|、^、&&、|| 中的任意一个，但使用减法操作会有一点问题，因为减法不满足交换律和结合律。例如，串行代码：

```
result = 0;
for (i = 1; i <= 4; i++)
   result -= i;
```

在 result 中存储的结果是 -10。然而，如果我们将迭代划分到两个线程中去执行，线程 0 将减 1 和 2，线程 1 将减 3 和 4，那么线程 0 将算出 -3，线程 1 将算出 -7。显然，-3-(-7)=4。[223] 理论上，编译器应该能指明线程各自的结果实际上应该要相加（-3+(-7)=-10），实际上，似乎也应该是这样。然而，OpenMP 标准 [42] 看来并不保证这一点。

还要注意，如果一个归约变量是一个 float 或 double 型数据，那么当使用不同数量的线程时，结果可能会有些许不同。这是由于浮点数运算不满足结合律。例如，如果 $a$、$b$ 和 $c$ 是浮点数，那么 $(a+b)+c$ 可能不会准确地等于 $a+(b+c)$。见习题 5.5。

当一个变量被包含在一个 reduction 子句中时，变量本身是共享的。然而，线程组中的每个线程都创建自己的私有变量。在 parallel 块里，每当一个线程执行涉及这个变量的语句时，它使用的其实是私有变量。当 parallel 块结束后，私有变量中的值被整合到一个共享变量中。因此，我们最新版本的代码是：

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

这段代码的执行效果与我们上个版本的代码（如下所示）相同。

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */
      my_result += Local_trap(double a, double b, int n);
#     pragma omp critical
      global_result += my_result;
   }
```

最后要注意的一点是，线程的私有变量初始化为0。这与初始化 my_result 为0是一样的。一般来说，根据不同的操作符，reduction 子句创建的私有变量初始化为相同的值。例如，如果操作符是乘法，则私有变量初始化为1。

## 5.5 parallel for 指令

作为梯形积分法显式并行化的替代方案，OpenMP 提供了 parallel for 指令。运用该指令，我们能够并行化串行梯形积分法：

224
```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```

方法是直接在 for 循环前放置一条指令：

```
   n = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```

与 parallel 指令一样，parallel for 指令生成一组线程来执行后面的结构化代码块。然而，在 parallel for 指令之后的结构化块必须是 for 循环。另外，运用 parallel for 指令，系统通过在线程间划分循环迭代来并行化 for 循环。因此，parallel for 指令与 parallel 指令非常不同，因为在 parallel 指令之前的块，一般来说其工作必须由线程本身在线程之间划分。

在一个已经被 parallel for 指令并行化的 for 循环中，线程间的缺省划分方式是由系统决定的。大部分系统会粗略地使用块划分，即如果有 $m$ 次迭代，则大约 $m$/thread_count 次迭代被分配到线程0，接下来的 $m$/thread_count 次被分配到线程1，以此类推。

注意，这里把 approx 作为一个归约变量是必要的。如果不那样做，它将是一个普通的共享变量，循环体中的

```
approx += f(a + i*h);
```

将会是一个无保护的临界区。

然而，说到作用域，在 parallel 指令中，所有变量的缺省作用域是共享的。但在 parallel for 中，如果循环变量 i 是共享的，那么变量更新 i++也会是一个无保护的临界区。因此，在一个被 parallel for 指令并行化的循环中，循环变量的缺省作用域是私有的，在我们的代码中，每个线程组中的线程拥有它自己的 i 的副本。

### 5.5.1 警告

这是一个十分美好的遐想：通过添加一条简单的 parallel for 指令，就可能并行化由大的 for 循环所组成的串行程序。也可能通过不断地在每个循环前放置 parallel for 指令，来增量地并行化一个串行程序。

然而，事情不会像看上去那样乐观。对 parallel for 的使用有几个警告。首先，OpenMP 只会并行化 for 循环，它不会并行化 while 或 do－while 循环。这似乎不是一个很大的限制，因
225 为任何使用 while 或 do－while 循环的代码都能够被转化为等效的使用 for 循环的代码。然而，OpenMP 只能并行化那些可以在如下情况下确定迭代次数的 for 循环：

- 由 for 语句本身（即 for（…；…；…））来确定。

- 在循环执行之前确定。

例如，“无限循环”：

```
for ( ; ; ) {
   . . .
}
```

不能被并行化。类似地，循环

```
for (i = 0; i < n; i++) {
   if ( . . . ) break;
   . . .
}
```

也不能被并行化，因为迭代的次数不能只从 for 语句中来决定。这个 for 循环也不是一个结构化块，因为 break 添加了另一个从循环退出的出口。

事实上，OpenMP 只能够并行化具有**典型结构**的 for 循环。典型的 for 循环采用程序 5-3 中的一种形式。这个模板中的变量和表达式符合一些十分明显的限制：

- 变量 index 必须是整型或指针类型（例如，它不能是 float 型浮点数）。
- 表达式 start、end 和 incr 必须有一个兼容的类型。例如，如果 index 是一个指针，那么 incr 必须是整型。
- 表达式 start、end 和 incr 不能够在循环执行期间改变。
- 在循环执行期间，变量 index 只能够被 for 语句中的“增量表达式”修改。

**程序 5-3　可并行化的 for 语句的合法表达形式**

```
                                  index++
                                  ++index
                    index < end   index--
                    index <= end  --index
for  index = start  ; index >= end ; index += incr
                    index > end   index -= incr
                                  index = index + incr
                                  index = incr + index
                                  index = index - incr
```

226

这些限制允许运行时系统在循环执行前确定迭代的次数。

运行时系统必须能够在执行前决定迭代的数量，但唯一例外的是：在循环体中可以有一个 exit 调用。

### 5.5.2　数据依赖性

如果 for 循环不能满足上述所列举规则中的任何一条，那么编译器将简单地拒绝它。例如，假设我们试图编译一个线性查找程序：

```
1     int Linear_search(int key, int A[], int n) {
2        int i;
3        /* thread_count is global */
4     #  pragma omp parallel for num_threads(thread_count)
5        for (i = 0; i < n; i++)
6           if (A[i] == key) return i;
7        return -1;  /* key not in list */
8     }
```

那么 gcc 编译器将报告：

```
Line 6: error: invalid exit from OpenMP structured block
```

一个更隐匿的问题发生在如下的循环中：在该循环中，迭代中的计算依赖于一个或更多个先

前的迭代结果。例如，考虑下述代码，计算前 $n$ 个斐波那契（Fibonacci）数：

```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
   fibo[i] = fibo[i-1] + fibo[i-2];
```

尽管怀疑其中有些问题，但我们还是用一个 parallel for 指令并行化这个 for 循环：

```
   fibo[0] = fibo[1] = 1;
#  pragma omp parallel for num_threads(thread_count)
   for (i = 2; i < n; i++)
      fibo[i] = fibo[i-1] + fibo[i-2];
```

编译器将创建一个可执行文件。然而，如果试图用多于一个线程去运行它，我们会发现结果是不可预计的。例如，在我们的一个系统上，如果试图使用两个线程去计算前 10 个斐波那契数，则我们有时会得到：

```
1    1    2    3    5    8    13    21    34    55
```

这是正确的。然而，我们也可能偶尔得到：

227

```
1    1    2    3    5    8    0    0    0    0
```

究竟发生了什么？似乎运行时系统将 fibo[2]、fibo[3]、fibo[4]和 fibo[5]的计算分配给了一个线程，而将 fibo[6]、fibo[7]、fibo[8]、fibo[9]分配给了另一个线程。（记住：循环从 i =2 开始。）在程序的一些运行结果中，结果之所以正确是因为被分配给 fibo[2]、fibo[3]、fibo[4]和 fibo[5]的线程在另一个线程开始前就完成了计算。然而，对于其他运行结果，当第二个线程计算 fibo[6]时，第一个线程还没有计算出 fibo[4]和 fibo[5]。系统将 fibo 的入口初始化为 0，第二个线程使用值 fibo[4] =0 和 fibo[5] =0 来计算 fibo[6]。然后它继续使用 fibo[5] =0 和 fibo[6] =0 来计算 fibo[7]，以此类推。

这里，我们看到两个要点：

（1）OpenMP 编译器不检查被 parallel for 指令并行化的循环所包含的迭代间的依赖关系，而是由程序员来识别这些依赖关系。

（2）一个或更多个迭代结果依赖于其他迭代的循环，一般不能被 OpenMP 正确地并行化。

fibo[6]和 fibo[5]计算间的依赖关系称为**数据依赖**。由于 fibo[5]的值在一个迭代中计算，其结果在之后的迭代中使用，该依赖关系有时称为**循环依赖**（loop-carried dependence）。

### 5.5.3 寻找循环依赖

当我们试图使用一个 parallel for 指令时，首先应该注意的是：要小心发现循环依赖。我们不需要担心一般的数据依赖。例如，在下列循环中：

```
1     for (i = 0; i < n; i++) {
2        x[i] = a + i*h;
3        y[i] = exp(x[i]);
4     }
```

在第 2 行和第 3 行之间有一个数据依赖。然而，如下的并行化没有问题。

```
1  #  pragma omp parallel for num_threads(thread_count)
2     for (i = 0; i < n; i++) {
3        x[i] = a + i*h;
4        y[i] = exp(x[i]);
5     }
```

因为 x[i]的计算与它接下来的使用总是被分配给同一个线程。

我们也应该观察到，有依赖关系的语句，其中至少一条语句会有序地写或更新变量。因此为

228 了检测循环依赖，我们只需要重点观察被循环体更新的变量，即我们应该寻找在一个迭代中被读

或被写，而在另一个迭代中被写的变量。我们来看几个例子。

### 5.5.4　π 值估计

一个对 π 的数值估计的方法是使用下列公式[⊖]：

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

我们能够在串行代码中实现这个公式。

```
1       double factor = 1.0;
2       double sum = 0.0;
3       for (k = 0; k < n; k++) {
4          sum += factor/(2*k+1);
5          factor = -factor;
6       }
7       pi_approx = 4.0*sum;
```

(factor 的类型是 double，而不是 int 或 long，为什么这点很重要?)

我们怎样用 OpenMP 来并行化它？我们可能首先倾向于这样做：

```
1       double factor = 1.0;
2       double sum = 0.0;
3  #    pragma omp parallel for num_threads(thread_count) \
4          reduction(+:sum)
5       for (k = 0; k < n; k++) {
6          sum += factor/(2*k+1);
7          factor = -factor;
8       }
9       pi_approx = 4.0*sum;
```

然而，可以清楚地看到，在第 k 次迭代中对第 7 行的 factor 的更新和接下来的第 k + 1 次迭代中对第 6 行的 sum 的累加是一个循环依赖。如果第 k 次迭代被分配给一个线程，而第 k + 1 次迭代被分配给另一个线程，则我们不能保证第 6 行中 factor 的值是正确的。在这种情况下，我们能通过检查系数来修复这个问题：

$$\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

229

我们可以看到：在第 $k$ 次迭代中，factor 的值应该是 $(-1)^k$。如果 $k$ 是偶数，那么值是 +1；如果 $k$ 是奇数，值是 -1。因此，如果将下述代码：

```
1          sum += factor/(2*k+1);
2          factor = -factor;
```

替换为：

```
1          if (k % 2 == 0)
2             factor = 1.0;
3          else
4             factor = -1.0;
5          sum += factor/(2*k+1);
```

或者，你可能更倾向于使用“?:”操作符：

```
1          factor = (k % 2 == 0) ? 1.0 : -1.0;
2          sum += factor/(2*k+1);
```

就消除了循环依赖性。

然而，事情仍然不是完全正确的。如果在我们的系统上使用两个线程运行程序，并设 $n$ =

---

⊖ 没有估计 π 值的最佳方法，因为它需要许多项来得到一个合理准确的结果。然而，我们对公式本身更感兴趣。

1000，那么结果仍是错误的。例如，

```
With n = 1000 terms and 2 threads,
   Our estimate of pi = 2.97063289263385
With n = 1000 terms and 2 threads,
   Our estimate of pi = 3.22392164798593
```

另一方面，如果只用一个线程运行程序，我们总是得到

```
With n = 1000 terms and 1 threads,
   Our estimate of pi = 3.14059265383979
```

到底哪里错了？

回想一下，在一个已经被 parallel for 指令并行化的块中，缺省情况下任何在循环前声明的变量（唯一的例外是循环变量）在线程间都是共享的。因此 factor 被共享。例如，线程 0 可能会给它赋值 1，但在它能用这个值更新 sum 前，线程 1 可能给它赋值 -1 了。因此，除了消除计算 factor 时的循环依赖外，我们还需要保证每个线程有它自己的 factor 副本。就是说，为了使代码正确，我们需要保证 factor 有私有作用域。通过添加一个 private 子句到 parallel 指令中来实现这一目标。

230

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) private(factor)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

在 private 子句内列举的变量，在每个线程上都有一个私有副本被创建。因此，在我们的例子中，thread_count 个线程中的每一个都有它自己的 factor 变量的副本，因此一个线程对 factor 的更新不会影响另一个线程的 factor 值。

要记住的重要的一点是，一个有私有作用域的变量的值在 parallel 块或者 parallel for 块的开始处是未指定的。它的值在 parallel 或 parallel for 块完成之后也是未指定的。例如，下列代码中的第一个 printf 语句的输出是非确定的，因为在它被显式初始化之前就打印了私有变量 x。类似地，最终的 printf 输出也是非确定的，因为它在 parallel 块完成之后打印 x。

```
   int x = 5;
#  pragma omp parallel num_threads(thread_count) \
      private(x)
   {
      int my_rank = omp_get_thread_num();
      printf("Thread %d > before initialization, x = %d\n",
            my_rank, x);
      x = 2*my_rank + 2;
      printf("Thread %d > after initialization, x = %d\n",
            my_rank, x);
   }
   printf("After parallel block, x = %d\n", x);
```

### 5.5.5 关于作用域的更多问题

关于变量 factor 的问题是常见问题中的一个。我们通常需要考虑在 parallel 块或 parallel for 块中的每个变量的作用域。因此，与其让 OpenMP 决定每个变量的作用域，还不如让程序员明确块中每个变量的作用域。事实上，OpenMP 提供了一个子句 default，该子句显式地

要求我们这样做。如果我们添加子句

```
default(none)
```

到 parallel 或 parallel for 指令中，那么编译器将要求我们明确在这个块中使用的每个变量和已经在块之外声明的变量的作用域。（在一个块中声明的变量都是私有的，因为它们会被分配给线程的栈。）

例如，使用一个 default(none)子句，对 π 的计算将如下所示。

```
      double sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         default(none) reduction(+:sum) private(k, factor) \
         shared(n)
      for (k = 0; k < n; k++) {
         if (k % 2 == 0)
            factor = 1.0;
         else
            factor = -1.0;
         sum += factor/(2*k+1);
      }
```

在这个例子中，我们在 for 循环中使用 4 个变量。由于 default 子句，我们需要明确每个变量的作用域。正如我们已经注意到的，sum 是一个归约变量（同时拥有私有和共享作用域的属性）。我们也已经注意到 factor 和循环变量 k 应该有私有作用域。从未在 parallel 或 parallel for 块中更新的变量，如这个例子中的 n，能够被安全地共享。与私有变量不同，共享变量在块内具有在 parallel 或 parallel for 块之前同样的值，在块之后的值与块内的最后一个值相同。因此，如果 n 在块之前被初始化为 1000，则在 parallel for 语句中它将保持这个值。因为在 for 循环中值没有改变，所以在循环结束后它将保持这个值。

## 5.6　更多关于 OpenMP 的循环：排序

### 5.6.1　冒泡排序

对一组整数排序的串行冒泡排序算法能够如下实现：

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

这里，数组 a 存储 $n$ 个 int 型整数，算法将它们升序排列。外循环首先找到列表的最大元素并将它存在 a[n-1]中，然后寻找次大的元素并存在 a[n-2]中，以此类推。因此，第一遍处理全部的 $n$ 个元素。第二遍处理除了最大元素外的所有元素，它处理一个 $n-1$ 个元素的列表，以此类推。

内循环比较当前列表中的连续元素对。当发现一对是无序的时候（a[i]>a[i+1]），就交换它们。这个交换过程将移动最大的元素到“当前”列表的最后，即由下列元素组成的列表：

```
a[0], a[1], . . . . , a[list_length-1]
```

显然，在外部循环中有一个循环依赖，在外部循环的任何一次迭代中，当前列表的内容依赖于外部循环的前一次迭代。例如，如果在算法开始时，a=3,4,1,2，那么外部循环的第二次迭代将对列表 3,1,2 进行操作，因为 4 在第一次迭代中应该已经被移动到列表的最后了。但如果前两次迭代同时执行，则有可能第二次迭代的有效列表包含 4。

内部循环的循环依赖也很容易发现。在第 $i$ 次迭代中，被比较的元素依赖于第 $i-1$ 次迭代。

如果在第 $i-1$ 次迭代中，a[i-1]和 a[i]没有交换，那么第 $i$ 次迭代将比较 a[i]和 a[i+1]。另一方面，如果第 $i-1$ 次迭代交换了 a[i-1]和 a[i]，那么第 $i$ 次迭代将比较原始的 a[i-1]（现在是 a[i]）和 a[i+1]。例如，假设当前列表是 {3,1,2}。那么当 $i=1$ 时，我们将比较 3 和 2，但如果 $i=0$ 和 $i=1$ 次迭代同时发生，则完全有可能 $i=1$ 次迭代会比较 1 和 2。

我们完全不清楚怎样在不完全重写算法的情况下移除任何一个循环依赖。记住，即使我们总能找到循环依赖，但可能很难甚至不可能移除它。对于并行化 for 循环而言，parallel for 指令不是一个通用的解决方案。

### 5.6.2 奇偶变换排序

奇偶变换排序是一个与冒泡排序相似的算法，但它相对来说更容易并行化。回想一下 3.7.1 节的串行奇偶排序，有如下实现：

```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n; i += 2)
         if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
   else
      for (i = 1; i < n-1; i += 2)
         if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

列表 a 存储 $n$ 个整数，算法对它们进行升序排列。在一个“偶阶段”（phase% 2 = =0）里，每个偶下标元素 a[i]与它左边的元素 a[i-1]相比较。如果它们是没有排好序的，就交换它们。在一个“奇阶段”里，每个奇下标元素与它右边的元素相比较。如果它们是没有排好序的，则交
233 换它们。有定理证明：在 $n$ 个阶段后，列表可以完成排序。

作为一个简单的例子，假设 a = {9,7,8,6}。表 5-1 显示了各个阶段的情况。在这个例子中，最后的阶段不是必要的，但算法并不在执行每个阶段前检查列表是否已经有序。

**表 5-1 串行的奇偶变换排序**

| 阶段 | 数组的下标 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | | | 1 | | | 2 | | | 3 |
| 0 | 9 | ↔ | | 7 | | | 8 | ↔ | | 6 |
| | 7 | | | 9 | | | 6 | | | 8 |
| 1 | 7 | | | 9 | ↔ | | 6 | | | 8 |
| | 7 | | | 6 | | | 9 | | | 8 |
| 2 | 7 | ↔ | | 6 | | | 9 | ↔ | | 8 |
| | 6 | | | 7 | | | 8 | | | 9 |
| 3 | 6 | | | 7 | ↔ | | 8 | | | 9 |
| | 6 | | | 7 | | | 8 | | | 9 |

不难看到外部循环有一个循环依赖。例如，假设在 a = {9,7,8,6} 之前。在阶段 0 中，内部循环将比较（9，7）和（8，6）这两对中的元素，这两对都会被交换。因此对于阶段 1，列表将是{7,9,6,8}，并在阶段 1 中（9，6）中的元素被比较并交换。然而，如果阶段 0 和阶段 1 同时执行，则在阶段 1 中被检查的可能是（7，8），是有序的。此外，我们尚不清楚如何消除这个循环依赖，因此并行化外部 for 循环不是一个好的选择。

但是，内部 for 循环并没有任何循环依赖。例如，在偶阶段循环中，变量 $i$ 是奇数，所以对于两个不同的 $i$ 值，例如，$i=j$ 和 $i=k$，$\{j-1, j\}$ 和 $\{k-1, k\}$ 将是不同的。（a[j-1],

a[j]) 和 (a[k-1], a[k]) 所产生的比较和可能的交换能够同时进行。

所以，我们试图使用程序 5-4 的代码并行化奇偶变换排序，但还是会有一些潜在的问题。首先，尽管任何一个偶阶段迭代并不依赖任何这个阶段的其他迭代，但是还需要注意，对 $p$ 阶段和 $p+1$ 阶段却不是这样的。我们需要确定在任何一个线程开始 $p+1$ 阶段之前，所有的线程必须先完成 $p$ 阶段。然而，像 parallel 指令那样，parallel for 指令在循环结束处有一个隐式的路障，因此，在所有的线程完成当前阶段（即阶段 $p$）之前，没有线程能够进入下一个阶段，即 $p+1$ 阶段。

其次，是创建和合并线程的开销。OpenMP 实现可能会在每一遍外部循环都创建和合并 thread_count 个线程。表 5-2 的第一行显示了当输入列表包含 20 000 个元素时，在我们系统上 234
运行 1、2、3、4 个线程的运行时间。

**程序 5-4　奇偶排序的第一个 OpenMP 实现**

```
1     for (phase = 0; phase < n; phase++) {
2        if (phase % 2 == 0)
3  #        pragma omp parallel for num_threads(thread_count) \
4              default(none) shared(a, n) private(i, tmp)
5           for (i = 1; i < n; i += 2) {
6              if (a[i-1] > a[i]) {
7                 tmp = a[i-1];
8                 a[i-1] = a[i];
9                 a[i] = tmp;
10             }
11          }
12       else
13 #        pragma omp parallel for num_threads(thread_count) \
14             default(none) shared(a, n) private(i, tmp)
15          for (i = 1; i < n-1; i += 2) {
16             if (a[i] > a[i+1]) {
17                tmp = a[i+1];
18                a[i+1] = a[i];
19                a[i] = tmp;
20             }
21          }
22    }
```

**表 5-2　用两条 parallel for 语句或两条 for 语句运行奇偶排序的时间（单位：秒）**

| thread_count | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 两条 parallel for 语句 | 0.770 | 0.453 | 0.358 | 0.305 |
| 两条 for 语句 | 0.732 | 0.376 | 0.294 | 0.239 |

这些时间耗费并不非常糟糕，但是我们想看看是否能做得更好。每次执行内部循环时，使用同样数量的线程。因此只创建一次线程，并在每次内部循环的执行中重用它们，这样做可能更好。幸运的是，OpenMP 提供了允许这样做的指令。用 parallel 指令在外部循环前创建 thread_count 个线程的集合。然后，我们不在每次内部循环执行时创建一组新的线程，而是使用一个 for 指令，告诉 OpenMP 用已有的线程组来并行化 for 循环。对原有 OpenMP 实现的改动显示在程序 5-5 中。

**程序 5-5　奇偶排序的第二个 OpenMP 实现**

```
1  #  pragma omp parallel num_threads(thread_count) \
2        default(none) shared(a, n) private(i, tmp, phase)
3     for (phase = 0; phase < n; phase++) {
```

```
4          if (phase % 2 == 0)
5  #          pragma omp for
6             for (i = 1; i < n; i += 2) {
7                if (a[i-1] > a[i]) {
8                   tmp = a[i-1];
9                   a[i-1] = a[i];
10                  a[i] = tmp;
11               }
12            }
13         else
14 #          pragma omp for
15            for (i = 1; i < n-1; i += 2) {
16               if (a[i] > a[i+1]) {
17                  tmp = a[i+1];
18                  a[i+1] = a[i];
19                  a[i] = tmp;
20               }
21            }
22      }
```

与 parallel for 指令不同的是，for 指令并不创建任何线程。它使用已经在 parallel 块中创
235 建的线程。在循环的末尾有一个隐式的路障。代码的结果（最终列表）将因此与原有的并行化代码所得到的结果一样。

奇偶排序的第二个版本的运行时间显示在表 5-2 的第二行。当使用两个或更多线程时，使用两条 for 指令的版本要比使用两条 parallel for 指令的版本快 17%。因此对于这个系统而言，为这点改变所做的小小努力是值得的。

## 5.7 循环调度

当第一次遇到 parallel for 指令时，我们看到将各次循环分配给线程的操作是由系统完成的。然而，大部分的 OpenMP 实现只是粗略地使用块分割：如果在串行循环中有 $n$ 次迭代，那么在并行循环中，前 $n$/thread_count 个迭代分配给线程 0，接下来的 $n$/thread_count 个迭代分配给线程 1，以此类推。不难想到，这种分配方式肯定不是最优的。例如，假设我们想要并行化循环：

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

236 同时，假设对 $f$ 函数调用所需要的时间与参数 $i$ 的大小成正比，那么与分配给线程 0 的工作相比，分配给线程 thread_count −1 的工作量相对较大。一个更好的分配方案是轮流分配线程的工作（循环划分）。在循环划分中，各次迭代被“轮流”地一次一个地分配给线程。假设 $t$ = thread_count。那么一个循环划分将如下分配各次迭代：

| 线程 | 迭代 | 线程 | 迭代 |
|---|---|---|---|
| 0 | 0, n=t, 2n=t, ⋯ | ⋮ | ⋮ |
| 1 | 1, n/t+1, 2n/t+1, ⋯ | t−1 | t−1, n/t+t−1, 2n/t+t−1, ⋯ |

为了了解这样的分配是如何影响性能的，我们编写了如下程序。

```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}  /* f */
```

每次函数$f(i)$调用$i$次sin函数。例如，执行$f(2i)$的时间几乎是执行$f(i)$的时间的两倍。

当$n=10\,000$并且只用一个线程运行程序时，运行时间是3.67秒。当用两个线程和缺省分配方式（第0～5000次迭代分配给线程0、第5001～10 000次迭代分配给线程1），运行程序时，运行时间是2.76秒。加速比仅为1.33。然而，当运行两个线程并采用循环划分时，运行时间减少到1.84秒。与单线程运行相比，加速比为1.99；与双线程、块分割相比，加速比为1.5！

我们看到一个好的迭代分配能够对性能有很大的影响。在OpenMP中，将循环分配给线程称为**调度**，schedule子句用于在parallel **for**或者**for**指令中进行迭代分配。

### 5.7.1 schedule子句

在例子中，我们已经知道如何获得缺省调度：只需要添加parallel **for**指令和reduction子句：

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
for (i = 0; i <= n; i++)
   sum += f(i);
```

237

为了对线程进行调度，可以添加一个schedule子句到parallel **for**指令中：

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

一般而言，schedule子句有如下形式：

```
schedule(<type> [, <chunksize>])
```

type可以是下列任意一个：

- static。迭代能够在循环执行前分配给线程。
- dynamic或guided。迭代在循环执行时被分配给线程，因此在一个线程完成了它的当前迭代集合后，它能从运行时系统中请求更多。
- auto。编译器和运行时系统决定调度方式。
- runtime。调度在运行时决定。

chunksize是一个正整数。在OpenMP中，迭代**块**是在顺序循环中连续执行的一块迭代语句，块中的迭代次数是chunksize。只有static、dynamic和guided调度有chunksize。这虽然决定了调度的细节，但准确的解释还是依赖于type。

### 5.7.2 static调度类型

对于static调度，系统以轮转的方式分配chunksize块个迭代给每个线程。例如，假设有12个迭代0、1、…、11和三个线程，如果在parallel **for**或**for**指令中使用schedule(**static**,1)，迭代将如下分配：

Thread 0：0，3，6，9
Thread 1：1，4，7，10
Thread 2：2，5，8，11

如果使用schedule(**static**,2)，迭代将如下分配：

238

Thread 0：0，1，6，7
Thread 1：2，3，8，9
Thread 2：4，5，10，11

如果使用 schedule(**static**,4)，迭代将如下分配：

Thread 0：0，1，2，3
Thread 1：4，5，6，7
Thread 2：8，9，10，11

因此，子句 schedule(**static**,total_iterations/thread_count)就相当于被大部分 OpenMP 实现所使用的缺省调度。

这里，chunksize 可以被忽略。如果它被忽略了，chunksize 就近似等于 total_iterations/thread_count。

### 5.7.3 dynamic 和 guided 调度类型

在 dynamic 调度中，迭代也被分成 chunksize 个连续迭代的块。每个线程执行一块，并且当一个线程完成一块时，它将从运行时系统请求另一块，直到所有的迭代完成。Chunksize 可以被忽略。当它被忽略时，chunksize 为 1。

在 guided 调度中，每个线程也执行一块，并且当一个线程完成一块时，将请求另一块。然而，在 guided 调度中，当块完成后，新块的大小会变小。例如，在我们的系统上，如果用 parallel **for** 指令和 schedule(guided)子句来运行梯形积分法程序，那么当 $n = 10\ 000$ 并且 thread_count = 2 时，迭代将如表 5-3 那样分配。块的大小近似等于剩下的迭代数除以线程数。第一块的大小为 9999/2≈5000，因为有 9999 个未被分配的迭代。第二块的大小为 4999/2≈2500，以此类推。

**表 5-3 使用 guided 调度为两个线程分配梯形积分法的 1～9999 次迭代**

| 线程 | 块 | 块的大小 | 剩下的迭代次数 |
|---|---|---|---|
| 0 | 1～5000 | 5000 | 4999 |
| 1 | 5001～7500 | 2500 | 2499 |
| 1 | 7501～8750 | 1250 | 1249 |
| 1 | 8751～9375 | 625 | 624 |
| 0 | 9376～9687 | 312 | 312 |
| 1 | 9688～9843 | 156 | 156 |
| 0 | 9844～9921 | 78 | 78 |
| 1 | 9922～9960 | 39 | 39 |
| 1 | 9961～9980 | 20 | 19 |
| 1 | 9981～9990 | 10 | 9 |
| 1 | 9991～9995 | 5 | 4 |
| 0 | 9996～9997 | 2 | 2 |
| 1 | 9998～9998 | 1 | 1 |
| 0 | 9999～9999 | 1 | 0 |

在 guided 调度中，如果没有指定 chunksize，那么块的大小为 1；如果指定了 chunksize，那么块的大小就是 chunksize，除了最后一块的大小可以比 chunksize 小。

### 5.7.4 runtime 调度类型

为了理解 schedule(runtime)，我们需要离题一会儿，讨论一下**环境变量**。正如名字所暗

示的，环境变量是能够被运行时系统所访问的命名值，即它们在程序的环境中是可得的。一些经常被使用的环境变量是 PATH、HOME 和 SHELL。PATH 变量明确了当寻找一个可执行文件时 shell 应该搜索哪些目录。它通常在 UNIX 和 Windows 系统中定义。HOME 变量指定用户主目录的位置，而 SHELL 变量指定用户 shell 的可执行位置。这些通常定义在 UNIX 系统中。在类 UNIX 系统（例如 Linux 和 Mac OS X）和 Windows 中，环境变量能够在命令行中检查和指定。在类 UNIX 系统中，239
能使用 shell 命令行；在 Windows 中，能使用集成开发环境的命令行。

例如，如果我们正使用 bash shell，要检查一个环境变量的值只需要输入

```
$ echo $PATH
```

我们能够使用 export 命令来设置一个环境变量的值

```
$ export TEST_VAR="hello"
```

如何检查和设置特定系统的环境变量，请咨询本地系统的专家。

当 schedule(runtime)指定时，系统使用环境变量 OMP_SCHEDULE 在运行时来决定如何调度循环。OMP_SCHEDULE 环境变量会呈现任何能被 static、dynamic 或 guided 调度所使用的值。例如，假设在程序中有一条 parallel **for** 指令，并且它已经被 schedule(runtime)修改了，那么如果使用 bash shell，就能通过执行以下命令将一个循环分配所得到的迭代分配给线程：

```
$ export OMP_SCHEDULE="static,1"
```

现在，当开始执行程序时，系统将调度 **for** 循环的迭代，就如同使用子句 schedule(**static**,1)修改了 parallel **for** 指令那样。240

### 5.7.5　调度选择

如果需要并行化一个 **for** 循环，那么我们如何决定使用哪一种调度和 chuncksize 的大小？实际上，每一种 schedule 子句有不同的系统开销。dynamic 调度的系统开销要大于 static 调度，而 guided 调度的系统开销是三种方式中最大的。因此，如果不使用 schedule 子句就已经达到了令人满意的性能，就不需要再进行多余的工作。但是，如果我们怀疑默认调度的性能可以提升，那么我们可以对各种调度进行试验。

在本节开始提供的例子中，在程序使用两个线程的情况下，使用 schedule(**static**,1)代替默认调度时，加速比从 1.33 增加到 1.99。因为在两个线程的条件下，加速比几乎不可能比 1.99 更好，所以我们可以不用再尝试其他的调度方式，至少在只用两个线程并且迭代数为 10 000 的情况下是这样。如果做更多的试验，改变线程的个数和迭代的次数，我们可能会发现：最优的调度方式是由线程的个数和迭代的次数共同决定的。

如果我们断定默认的调度方式性能低下，那么我们会做大量的试验来寻找最优的调度方式和迭代次数。在进行了大量的工作以后，我们可能发现，这些循环没有得到很好的并行化，没有哪一种调度可以带来比较显著的性能提升。编程作业 5.4 就是这样的一个例子。

但在某些情况下，应该优先考虑有些调度：

- 如果循环的每次迭代需要几乎相同的计算量，那么可能默认的调度方式能提供最好的性能。
- 如果随着循环的进行，迭代的计算量线性递增（或者递减），那么采用比较小的 chuncksize 的 static 调度可能会提供最好的性能。
- 如果每次迭代的开销事先不能确定，那么就可能需要尝试使用多种不同的调度策略。在这种情况下，应当使用 schedule(runtime)子句，通过赋予环境变量 OMP_SCHEDULE 不同的值来比较不同调度策略下程序的性能。

## 5.8 生产者和消费者问题

本节将讨论一个不适合用 parallel for 指令或者 for 指令来并行化的问题。

### 5.8.1 队列

241 **队列**是一种抽象的数据结构，插入元素时将元素插入到队列的“尾部”，而读取元素时，队列“头部”的元素被返回并从队列中被移除。队列可以看做是在超市中等待付款的消费者的抽象，队列中的元素是消费者。新的消费者到达时排在等待队列的尾部，下一个付款离开等待队列的是排在队列头部的消费者。

当一个新的元素插入到队列的尾部时，通常称这个新的元素“入队”了；当一个元素从队列的头部被移除时，通常称这个元素“出队”了。

队列在计算机科学中随处可见。例如，如果有多个进程，每个进程都试图向硬盘写入数据，为了确保每次只有一个进程在写硬盘，一种自然而然的方法是将进程组织为队列。换句话说，排在队列第一个的进程在当前进程结束对硬盘的使用后，第一个获得硬盘的访问权限；排在队列第二个的进程在排在队列第一个的进程使用完硬盘后获得硬盘的访问权限，以此类推。

队列也是在多线程应用程序中经常使用到的数据结构。例如，我们有几个“生产者”线程和几个“消费者”线程。生产者线程“产生”对服务器数据的请求——例如当前股票的价格，而消费者线程通过发现和生成数据（例如，当前股票的价格）来“消费”请求。生产者线程将请求入队，而消费者线程将请求从队列中移出。在这个例子中，只有当消费者线程将请求的数据发送给生产者线程时，进程才会结束。

### 5.8.2 消息传递

生产者和消费者问题模型的另外一个应用是在共享内存系统上实现消息传递。每一个线程有一个共享消息队列，当一个线程要向另外一个线程“发送消息”时，它将消息放入目标线程的消息队列中。一个线程接收消息时只需从它的消息队列的头部取出消息。

这里我们将实现一个简单的消息传递程序，在这个程序中，每个线程随机产生整数“消息”和消息的目标线程。当创建一条消息后，线程将消息加入到合适的消息队列中。当发送消息之后，该线程查看它自己的消息队列以获知它是否收到了消息，如果它收到了消息，它将队首的消息出队并打印该消息。每个线程交替发送和接收消息，用户需要指定每个线程发送消息的数目。当一个线程发送完所有的消息后，该线程不断接收消息直到其他所有的线程都已完成，此时所有的线程都结束了。每个线程的伪代码如下：

242
```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
   Send_msg();
   Try_receive();
}

while (!Done())
   Try_receive();
```

### 5.8.3 发送消息

需要注意的是，访问消息队列并将消息入队，可能是一个临界区。尽管我们还没有深入地研究如何实现消息队列，但我们很有可能需要用一个变量来跟踪队列的尾部。例如，使用一个单链表来实现消息队列，链表的尾部对应着队列的尾部。然后，为了有效地进行入队操作，需要存储指向链表尾部的指针。当一条新消息入队时，需要检查和更新这个队尾指针。如果两个线程试图同时进行这些操作，那么可能会丢失一条已经由其中一个线程入队的消息。（画张图能够有助于

理解这种情况!）两个操作的结果会发生冲突，因此入队操作形成了临界区。

Send_msg()函数的伪代码如下：

```
   mesg = random();
   dest = random() % thread_count;
#  pragma omp critical
   Enqueue(queue, dest, my_rank, mesg);
```

注意在上面的实现中，允许线程向它自己发送消息。

### 5.8.4 接收消息

接收消息的同步问题与发送消息有些不同。只有消息队列的拥有者（即目标线程）可以从给定的消息队列中获取消息。如果消息队列中至少有两条消息，那么只要每次只出队一条消息，那么出队操作和入队操作就不可能冲突。因此如果队列中至少有两条消息，通过跟踪队列的大小就可以避免任何同步（例如 critical 指令）。

现在的问题是如何存储队列大小。如果只使用一个变量来存储队列的大小，那么对该变量的操作会形成临界区。然而可以使用两个变量：enqueued 和 dequeued，那么队列中消息的个数（队列的大小）就为

```
queue_size = enqueued - dequeued
```

并且，唯一能够更新 dequeued 的线程是消息队列的拥有者。可以看到在一个线程使用 enqueued 计算队列大小 queue_size 的同时，另外一个线程可以更新 enqueued。为了解释这种情况，假设进程 $q$ 正在计算 queue_size，那么它将可能得到 enqueued 新的或者旧的值。当 queue_size 实际的值是 1 或者 2 时，线程 $q$ 可能会得到 queue_size 是 0 或者 1。但这只会引起程序一定的延迟，而不会引起程序错误。如果 queue_size 本应该是 1，却误计算为 0，那么线程 $q$ 延迟一段时间后会试图重新计算队列的大小；如果 queue_size 本应该是 2，却误计算为 1，那么线程 $q$ 将执行临界区指令，虽然这本来是不必要的。

因此，可以按照如下的方式实现 Try_receive：

```
queue_size = enqueued - dequeued;
if (queue_size == 0) return;
   else if (queue_size == 1)
#     pragma omp critical
      Dequeue(queue, &src, &mesg);
   else
      Dequeue(queue, &src, &mesg);
   Print_message(src, mesg);
```

243

### 5.8.5 终止检测

接下来，我们探讨如何实现 Done 函数。首先，我们给出一个“直接”的实现，但这个实现隐藏着问题：

```
queue_size = enqueued - dequeued;
if (queue_size == 0)
   return TRUE;
else
   return FALSE;
```

如果线程 $u$ 执行这段代码，那么很有可能有些线程，如线程 $v$，在线程 $u$ 计算出 queue_size = 0 后向线程 $u$ 发送一条消息。当然，线程 $u$ 在得出 queue_size = 0 后将终止，那么线程 $v$ 发送给它的消息就永远不会被接收到。

然而，在我们的程序中，每个线程在执行完 for 循环后将不再发送任何消息。因此可以增加

一个计数器 done_sending，每个线程在 for 循环结束后将该计数器加 1，Done 的实现如下：

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
   return TRUE;
else
   return FALSE;
```

### 5.8.6 启动

当程序开始执行时，主线程将得到命令行参数并且分配一个数组空间给消息队列，每个线程对应着一个消息队列。由于每个线程可以向其他任意的线程发送消息，所以这个数组应该被所有的线程共享，而且每个线程可以向任何一个消息队列插入一条消息。消息队列（至少）可以存储：

- 消息列表
- 队尾指针或索引
- 队首指针或索引
- 入队消息的数目
- 出队消息的数目

最好将队列存在消息队列的结构体中，为了减少参数传递时复制的开销，最好用指向结构体
244 的指针数组来实现消息队列。因此，一旦主线程分配了队列数组，就可以使用 parallel 指令开始执行线程，每个线程可以为自己的队列分配存储空间。

这里一个重要的问题是：一个或者多个线程可能在其他线程之前完成它的队列分配。如果这种情况出现了，那么完成分配的线程可能会试图开始向那些还没有完成队列分配的线程发送消息，这将导致程序崩溃。因此，我们必须确保任何一个线程都必须在所有的线程都完成了队列分配后才开始发送消息。回想一下，之前我们见过一些 OpenMP 指令在结束时提供隐式路障，即任何一个线程都必须等到组中所有的线程完成了某个程序块后才可以接着执行后续代码。然而，在这个例子中，我们处于 parallel 块的中间，所以我们不能依赖于 OpenMP 提供的隐式路障——我们应当使用显式的路障。幸运的是，OpenMP 提供了相应的指令：

```
# pragma omp barrier
```

当线程遇到路障时，它将被阻塞，直到组中所有的线程都到达了这个路障。当组中所有的线程都到达了这个路障时，这些线程就可以接着往下执行。

### 5.8.7 atomic 指令

发送完所有的消息后，每个线程在执行最后的循环以便接收消息之前，需要对 done_sending 加 1。显然，对 done_sending 的增量操作是临界区，可以通过 critical 指令来保护它。然而，OpenMP 提供了另外一种可能更加高效的指令：atomic 指令：

```
# pragma omp atomic
```

与 critical 指令不同，它只能保护由一条 C 语言赋值语句所形成的临界区。此外，语句必须是以下几种形式之一：

```
x <op>= <expression>;
x++;
++x;
x—;
—x;
```

< op > 可以是以下任意的二元操作符：

+, *, -, /, &, ^, |, <<, or >>.

这里要记住，<expression>不能引用 x。

需要注意的是，只有 x 的装载和存储可以确保是受保护的，例如在下面的代码中：

```
#     pragma omp atomic
      x += y++;
```

245

其他线程对 x 的更新必须等到该线程对 x 的更新结束之后。但是对 y 的更新不受保护，因此程序的结果是不可预测的。

atomic 指令的思想是许多处理器提供专门的装载 - 修改 - 存储（load - modify - store）指令。使用这种专门的指令而不使用保护临界区的通用结构，可以更高效地保护临界区。

### 5.8.8 临界区和锁

为了完成对消息传递程序的讨论，我们需要进一步仔细研究 OpenMP critical 指令的规范。在更早的例子中，程序最多只有一个临界区，critical 指令强制所有的线程对该区域进行互斥访问。在这个程序中，临界区的使用将更加复杂。我们将在源代码中看到3个在 critical 或 atomic 指令后面的代码块：

- done_sending++
- Enqueue(q_p,my_rank,mesg);
- Dequeue(q_p,&src,&mesg);

然而，我们不需要强制对3个代码块都进行互斥访问，甚至不需要强制对第2个和第3个代码块进行完全的互斥访问。例如，线程0在向线程1的消息队列写消息的同时，线程1可以向线程2的消息队列写消息。但是，根据 OpenMP 的规定，第2个和第3个代码块是被 critical 指令保护的代码块。在 OpenMP 看来，我们的程序有两个不同的临界区：被 atomic 指令保护的 done_sending++和“复合”临界区。在“复合”临界区中，程序读取和发送消息。

强制线程间的互斥会使程序的执行串行化。OpenMP 默认的做法是将所有的临界区代码块作为复合临界区的一部分，这可能非常不利于程序的性能。OpenMP 提供了向 critical 指令添加名字的选项：

```
# pragma omp critical(name)
```

采取这种方式，两个用不同名字的 critical 指令保护的代码块就可以同时执行。我们想为每一个线程的消息队列的临界区提供不同的名字，但是临界区的名字是在程序编译过程中设置的。因此，我们需要在程序执行的过程中设置临界区的名字。但是按照我们的设置，当我们想让访问不同队列的线程可以同时访问相同的代码块时，被命名的 critical 指令就不能满足我们的要求了。

246

解决方案是使用锁（lock）[⊖]。锁由一个数据结构和定义在这个数据结构上的函数组成，这些函数使得程序员可以显式地强制对临界区进行互斥访问。锁的使用可以大概用下面的伪代码描述：

```
/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;
```

⊖ 如果你已经阅读过 Pthreads 的章节，那么你已经了解锁机制，可以跳到后面 OpenMP 锁的语法部分。

锁的数据结构被执行临界区的线程所共享，这些线程中的某个线程（如主线程）会初始化锁。而当所有的线程都使用完锁后，某个线程应当负责销毁锁。

在一个线程进入临界区前，它尝试通过调用锁函数来上锁（set）。如果没有其他的线程正在执行临界区的代码，那么它将获得锁并进入临界区。当该线程执行完临界区的代码后，它调用解锁函数释放（relinquish 或者 unset）锁，以便其他线程可以获得锁。

当一个线程拥有锁时，其他的线程都不能进入该临界区。其他线程尝试通过调用锁函数进入该临界区时会被阻塞。如果有多个线程被锁函数阻塞，则当临界区的线程释放锁时，这些线程中的某个线程会获得锁，而其他线程仍被阻塞。

OpenMP 有两种锁：**简单**（simple）锁和**嵌套**（nested）锁。简单锁在被释放前只能获得一次，而一个嵌套锁在被释放前可以被同一个线程获得多次。OpenMP 简单锁的类型是 omp_lock_t，定义简单锁的函数包括：

```
void omp_init_lock(omp_lock_t*     lock_p   /* out */);
void omp_set_lock(omp_lock_t*      lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*    lock_p   /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p    /* in/out */);
```

相关的类型和函数在头文件 omp.h 中声明。第一个函数的作用是初始化锁，所以此时锁处于解锁状态，换句话说，此时没有线程拥有这个锁。第二个函数尝试获得锁，如果成功，调用该函数
247 的线程可以继续执行；如果失败，调用该函数的线程将被阻塞，直到锁被其他线程释放。第三个函数释放锁，以便其他线程可以获得该锁。第四个函数销毁锁。本书仅涉及简单锁，如果想了解嵌套锁的知识，可以查看［8，10］或者［42］。

### 5.8.9 在消息传递程序中使用锁

在前面对 critical 指令不足之处的讨论中，我们看到，在消息传递程序中，我们想要确保的是对每个消息队列进行互斥访问，而不是对于一个特定的代码块。锁可以帮助我们实现这个目的。将 omp_lock_t 类型的数据成员包含在队列结构中，可以通过简单地调用 omp_set_lock 函数来确保对消息队列的互斥访问。所以代码：

```
#  pragma omp critical
   /* q_p = msg_queues[dest] */
   Enqueue(q_p, my_rank, mesg);
```

可以用以下代码代替：

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

类似地，代码：

```
#  pragma omp critical
   /* q_p = msg_queues[my_rank] */
   Dequeue(q_p, &src, &mesg);
```

可以用以下代码代替：

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

现在，当一个线程试图发送或者接收消息时，它只可能被其他试图访问相同消息队列的线程阻塞，因为每一个消息队列拥有不同的锁。在我们最初的实现中，不管消息的目的地是哪个队

列，每次都只有一个线程可以发送消息。

需要注意的是，对锁函数的调用也可以放在队列函数 Enqueue 和 Dequeue 中。但是，为了确保 Dequeue 函数的性能，还需要将判断队列大小的代码(enqueued - dequeued)放到 Dequeue 函数中。如果不这样做，每次被 Try_receive 函数调用时，Dequeue 函数都会锁住整个队列。为了保持已写代码的结构，我们将对 omp_set_lock 和 omp_unset_lock 的调用放在 Send 和 Try_receive 函数中。

因为我们已经将队列相应的锁包含在了队列的结构中，所以我们可以把对锁初始化的代码添加到初始化空队列的函数中，而对锁的销毁则可以由拥有该队列的线程在销毁队列前完成。 248

### 5.8.10 critical 指令、atomic 指令、锁的比较

到目前为止，有三种机制可以实现对临界区的访问，很自然地我们想知道在不同的情况下应当采取哪一种方法。一般而言，atomic 指令是实现互斥访问最快的方法。因此，如果临界区由特定形式的赋值语句组成，则使用 atomic 指令至少不会比使用其他方法慢。然而，OpenMP 规范[42]允许 atomic 指令对程序中所有 atomic 指令标记的临界区进行强制互斥访问——这是未命名的 critical 指令的工作方式。如果这种行为不能满足要求（例如，程序中有多个不同的由 atomic 指令保护的临界区）则应当使用命名的 critical 指令或者锁。例如，假设在程序中有一个线程执行下面左边的代码，而另外一个线程执行右边的代码：

```
# pragma omp atomic          # pragma omp atomic
  x++;                         y++;
```

即使 x 和 y 拥有不同的内存地址，但可能一个线程在执行 x++操作时，其他线程不能同时执行 y++。需要注意的是，这种行为其实是不必要的。如果两条修改不同变量的语句都被 atomic 指令保护，那么有些 OpenMP 实现会把这两条语句视为不同的临界区，具体参见习题 5.10。另一方面，不管语句以何种形式实现，修改同一变量的不同语句都将视为属于同一个临界区。

前面我们已经看到了一些使用 critical 指令的不足之处。然而，不论是未命名的 critical 指令还是命名的 critical 指令都十分易于使用。此外，在我们所使用到的 OpenMP 的实现中，使用 critical 指令保护临界区与使用锁保护临界区在性能上没有太大的差别。所以，如果我们在无法使用 atomic 指令，却能够使用 critical 指令时，我们应当使用 critical 指令。锁机制适用于需要互斥的是某个数据结构而不是代码块的情况。

### 5.8.11 经验

在使用我们讨论过的这些互斥技术时应当谨慎，它们肯定会引起一些严重的编程问题，下面是一些你需要知道的要点：

(1) 对同一个临界区不应当混合使用不同的互斥机制。例如，一个程序包含下面的两个代码片段：

```
# pragma omp atomic          # pragma omp critical
  x += f(y);                   x = g(x);
```

249

右边的代码对 x 进行修改，但是不满足 atomic 指令要求的形式，所以程序员采用了 critical 指令。由于 critical 指令和 atomic 指令不会互斥执行，所以程序可能会得到不正确的结果。程序员需要重写函数 $g$，使得它满足 atomic 指令要求的形式，或者程序员需要用 critical 指令将这两个代码块都保护起来。

(2) 互斥执行不保证**公平性**，也就是说可能某个线程会被一直阻塞以等待对某个临界区的执行。例如下面的代码中：

```
   while(1) {
      . . .
#     pragma omp critical
      x = g(my_rank);
      . . .
   }
```

线程 1 可能被一直阻塞以等待执行 x = g(my_rank)，而其他的线程却在反复执行这条赋值语句。当然，如果循环能结束，那么上面的问题就不会存在。另外在许多实现中，线程按照到达的先后顺序进入临界区，在这种情况下，上面的问题也不会存在。

(3) “嵌套”互斥结构可能会产生意料不到的结果。例如，一个程序包含了以下的两个代码片段：

```
#  pragma omp critical
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical
      z = g(x);  /* z is shared */
      . . .
   }
```

这段代码肯定会产生**死锁**。当一个线程试图进入第二个临界区时，它将被永远阻塞。如果一个线程 $u$ 在执行第一个临界区中的代码，则不可能有其他的线程执行第二个临界区中的代码。然而，如果线程 $u$ 被阻塞以等待进入第二个临界区，那么它将永远不会离开第一个临界区，换句话说，它将被永远阻塞。

在这个例子中，我们使用命名临界区来解决上面的问题。我们重写代码如下：

250

```
#  pragma omp critical(one)
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical(two)
      z = g(x);  /* z is global */
      . . .
   }
```

然而，还是有命名临界区不能解决的问题。例如，如果一个程序有两个命名临界区（one 和 two）并且线程试图以不同的顺序进入临界区，那么死锁就可能发生。例如，假设线程 $u$ 进入临界区 one 的同时线程 $v$ 进入临界区 two，然后线程 $u$ 试图进入临界区 two 的同时线程 $v$ 试图进入临界区 one：

| 时间 | 线程 $u$ | 线程 $v$ |
|---|---|---|
| 0 | 进入临界区 one | 进入临界区 two |
| 1 | 试图进入临界区 two | 试图进入临界区 one |
| 2 | 阻塞 | 阻塞 |

那么，线程 $u$ 和 $v$ 都将永远被阻塞以等待进入临界区。由此可见，仅仅使用命名临界区不足以解决上面的问题——程序员必须确保各个线程以相同的顺序进入临界区。

## 5.9 缓存、缓存一致性、伪共享[⊖]

多年来，处理器的执行速度已经远远超过了它们从主存中访问数据的速度，所以如果处理器

⊖ 这部分的内容在第 4 章已经介绍过。如果你已经读过，那么可以跳过这部分内容。

的每一个操作都要从主存中访问数据，那么它们大多数时间都是在等待数据从主存中到达。为了解决这个问题，处理器设计人员在处理器中加入了比主存更快的存储器——**缓存**（cache）。

缓存的设计考虑到了*时间和空间局部性*：如果一个处理器在时间 $t$ 访问了主存地址 $x$，那么很有可能它会在与 $t$ 相近的时间内访问主存中靠近 $x$ 的地址。因此，如果处理器要访问主存地址 $x$，那么一个包含 $x$ 中内容的存储块将被写入缓存中或者从缓存中读出，而不是仅仅将 $x$ 中的内容写入缓存或者从缓存中读出。这样的存储块叫做**缓存行**或者**缓存块**。

我们已经在 2.3.4 节看到缓存的使用对于共享内存有着巨大的影响。让我们回忆一下这是为什么。首先，考虑下面的情况。假设共享变量 $x$ 的值为 5，因为要执行下面的代码

```
my_y = x;
```

251

所以线程 0 和线程 1 都将分别从内存中将 $x$ 读入它们的缓存。在这里，my_y 是分别定义在两个线程内部的私有变量。假设线程 0 执行下面的语句：

```
x++;
```

最后，假设线程 1 现在要执行：

```
my_z = x;
```

my_z 是定义在线程内部的另一个私有变量。

现在 my_z 是多少？5 还是 6？现在的问题是（至少）存在 3 个 x 的副本：一个在主存中，一个在线程 0 的缓存中，一个在线程 1 的缓存中。当线程 0 执行 x++时，主存中和线程 1 的缓存中的 x 应当如何变化呢？这就是第 2 章讨论的缓存一致性问题。我们看到大部分系统都坚持缓存能获知它们所缓存的数据的改变。当线程 0 执行 x++时，线程 1 中 x 所在的缓存块被标记为 *invalid*。在执行赋值语句 my_z = x 之前，运行线程 1 的核将会获知 x 的值已经过期了。因此运行线程 0 的核必须更新 x 在内存中的副本（现在或者更早一些），运行线程 1 的核将从主存中获取包含更新过的 x 的内存块。详见第 2 章。

缓存一致性的使用对共享内存系统的性能有着巨大的影响。为了说明这点，我们来看一看矩阵 - 向量乘法的例子。假设 $\boldsymbol{A}=(a_{ij})$ 是 $m\times n$ 矩阵，$\boldsymbol{x}$ 是有 $n$ 个元素的向量，它们的积 $\boldsymbol{y}=\boldsymbol{A}\boldsymbol{x}$ 是一个有 $m$ 个元素的向量，并且第 $i$ 个元素 $y_i$ 由 $\boldsymbol{A}$ 的第 $i$ 行和 $\boldsymbol{x}$ 的内积得到：

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

见图 5-5。

所以，如果将 $\boldsymbol{A}$ 存储为二维数组，$\boldsymbol{x}$ 和 $\boldsymbol{y}$ 存储为一维数组，则可以使用下面的代码实现串行化的矩阵 - 向量乘法：

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

图 5-5　矩阵 - 向量乘法

252

因为 A 和 x 都没有改变，且第 i 次迭代只更新变量 y[i]，所以在上面的代码中外循环没有循环依赖。通过将外循环中的迭代划分给不同的线程，可以将上面的代码并行化：

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

设 $T_{串行}$ 是串行程序的运行时间，$T_{并行}$ 是并行程序的运行时间，并行程序的效率 $E$ 是加速比除以线程的数目 $t$：

$$E = \frac{S}{t} = \frac{\left(\frac{T_{串行}}{T_{并行}}\right)}{t} = \frac{T_{串行}}{t \times T_{并行}}$$

$S \leqslant t$，$E \leqslant 1$。表 5-4 展示了我们的矩阵向量乘法在不同数据集和线程数下的运行时间和效率。

表 5-4　矩阵－向量乘法的运行时间（单位：秒）和效率

| 线程 | 矩阵的维度 | | | | | |
|---|---|---|---|---|---|---|
| | 8 000 000 ×8 | | 8000 ×8000 | | 8 ×8 000 000 | |
| | 时间 | 效率 | 时间 | 效率 | 时间 | 效率 |
| 1 | 0.322 | 1.000 | 0.264 | 1.000 | 0.333 | 1.000 |
| 2 | 0.219 | 0.735 | 0.189 | 0.698 | 0.300 | 0.555 |
| 4 | 0.141 | 0.571 | 0.119 | 0.555 | 0.303 | 0.275 |

在每种情况下，浮点数加法与乘法操作的总数都是 64 000 000 次。只考虑算术运算次数的分析预测单线程运行程序在三种输入方式情况下的运行时间是相同的。然而，这与我们看到的实际情况并不相同。对于 8 000 000 ×8 这一输入，比起 8000 ×8000 而言，系统需要 22% 的多余时间。而若输入为 8 ×8 000 000，则需要 26% 的多余时间。这两处的不同运行时间，至少部分原因是由缓存性能引起的。

当核试图修改不在缓存中的变量时，会发生**写缺失**（write-miss），此时内核必须访问主存。缓存分析器（Valgrind [49]）表明当程序的输入是 8 000 000 ×8 时，比其他输入产生更多的写缺
253 失，并且大多数的写缺失发生在第 4 行。因为在这种情况下，向量 y 中元素的数目非常大（8 000 000 与 8 000 或 8），且每一个元素都必须初始化，那么第 4 行代码在输入为 8 000 000 ×8 时执行速度减慢就在情理之中了。

当核试图读取不在缓存中的变量时，会发生**读缺失**（read-miss），此时它也必须访问主存。缓存分析器表明在输入为 8 ×8 000 000 时，比其他输入产生更多的读缺失。读缺失发生在程序的第 6 行，仔细研究程序（见习题 5.12）会发现，这些不同的主要原因是对变量 x 的读取。对于输入 8 ×8 000 000 而言，x 有 8 000 000 个元素，而对于其他输入只有 8000 个或者 8 个元素，这就造成了读缺失次数的不同。

另外需要谨记的是：存在其他原因影响单线程程序在不同输入下的性能。例如，我们没有考虑虚拟内存（见 2.2.4 节）是否影响程序在不同输入下的执行，以及 CPU 访问主存中页表的频率。

我们最感兴趣的是当线程数目增加时程序性能的变化。对于双线程的程序，程序在输入为 8 ×8 000 000 时比在输入为 8000 ×8000 和 8 000 000 ×8 时性能至少下降了 20%。对于 4 个线程的

程序，程序在输入为 8 × 8 000 000 时比在输入为 8000 × 8000 和 8 000 000 × 8 时性能至少下降了 50%。那么，为什么在输入为 8 000 000 × 8 时，多线程程序的性能会这么差呢？

同样，这个现象与缓存有关。让我们仔细研究 4 个线程的程序。在输入为 8 000 000 × 8 时，y 有 8 000 000 个元素，所以给每个线程分配了 2 000 000 个元素；在输入为 8000 × 8000 时，给每个线程分配了 y 中的 2000 个元素；在输入为 8 × 8 000 000 时，给每个线程分配 y 中的 2 个元素。在我们的系统中，一个缓存块有 64 字节，由于 y 是双精度浮点型，一个双精度浮点变量占 8 字节，所以一个缓存块可以储存 8 个双精度浮点变量。

缓存一致性是在“缓存行级别”上执行的，换句话说，只要缓存行中的某个变量改变了，且其他核的缓存也存储了该变量，那么将整个缓存行标记为不合法——不只是被改变的变量。我们使用的系统有 2 个双核的处理器，每个处理器都有自己的缓存。假设此时线程 0 和线程 1 分配给了其中的一个处理器，线程 2 和线程 3 分配给了另外一个处理器。还假设在输入为 8 × 8 000 000 时，y 的所有元素都存储在一个缓存行中，那么每次对 y 中元素的写都将导致其他处理器中的缓存行失效。例如，每一次线程 0 在下面的语句中更新 y[0]

```
y[i] += A[i][j]*x[j];
```

如果线程 2 或者线程 3 正在执行程序，那么它们必须重新载入 y 的值。每一个线程都要更新它自己的元素 8 000 000 次。我们看到，由于这条赋值语句，所有的线程都必须重新载入 y 很多次。254
只要有一个线程访问 y 中的任意一个元素，上述情况就会发生——比如只有线程 0 访问 y[0]。

每个线程要更新分配的 y 中的元素 16 000 000 次，并且其中的大多数更新都会迫使线程访问主存，这种情况称为**伪共享**。假设有着不同缓存的两个线程访问同一个缓存行中的不同变量，还假设至少有一个线程更新了变量的值。尽管没有线程对共享变量进行了更新，但缓存控制器会将整个缓存行失效，从而迫使其他线程从主存中获取变量的值。这些线程之间没有共享任何变量（只是共享了同一个缓存行），但是它们访问主存的行为看起来好像它们共享了一个变量，所以这种情况称为*伪共享*。

为什么在其他输入下，伪共享不会带来问题呢？让我们看看在输入为 8000 × 8000 时程序运行的情况。假定将线程 2 分配给了其中的一个处理器，而线程 3 分配给了另外一个处理器（我们不能准确地知道线程分配给了哪个处理器，但是结果表明（见练习 5.13）这不会对结果产生影响）。线程 2 负责计算

```
y[4000], y[4001], . . . , y[5999]
```

而线程 3 负责计算

```
y[6000], y[6001], . . . , y[7999]
```

如果一个缓存块包含 8 个连续的 `double` 型变量，那么伪共享只可能发生在元素的连接处。例如，假设一个缓存块包含

```
y[5996], y[5997], y[5998], y[5999], y[6000], y[6001], y[6002], y[6003]
```

那么可以预见这个缓存块可能发生伪共享。然而，线程 2 在 `for` 循环的迭代结束时访问

```
y[5996], y[5997], y[5998], y[5999]
```

而线程 3 在迭代开始时访问

```
y[6000], y[6001], y[6002], y[6003]
```

所以很有可能在线程 2 访问，比如说，y[5996]时，线程 3 早已经完成了对

```
y[6000], y[6001], y[6002], y[6003]
```

的计算。同样，当线程3访问，比如说，y[6003]时，线程2很可能将不会访问

255 y[6000], y[6001], y[6002], y[6003]

因此在输入为8000×8000时，y元素的伪共享可能不会带来较严重的问题。同样道理，在输入为8 000 000×8时，y的伪共享也可能不会造成太大的问题。需要注意的是，不需要担心对A和x的伪共享，因为它们的值从来没有被矩阵-向量乘法程序更新过。

上面的内容引出了一个问题：我们应当怎样避免矩阵-向量乘法程序的伪共享呢？一种可行的解决方案是用伪变量填充向量y，以确保其中任意一个线程的更新不会影响其他线程的缓存行。另外一个可行的解决方案是每个线程在迭代期间使用私有存储，然后在计算完成后更新共享存储（见习题5.15）。

## 5.10 线程安全性[⊖]

本节将探讨在共享内存编程中另一个可能发生的问题：*线程安全性*。当一个代码块被多个线程同时执行时不会产生错误，那么这个代码块是**线程安全的**。

为了举例说明，假设要使用多个线程对一个文件进行“分词”。假定这个文件由普通的英文文本组成，每个单词是一个连续的字母序列且用空白符（空格、制表符，或者换行符）与其他的文本隔开。一个简单的方法是按行对输入文件进行划分，然后以轮转的方式将文本行分配给各个线程：第1行分配给线程0，第2行分配给线程1，…，第$t$行分配给线程$t-1$，第$t+1$行分配给线程0，以此类推。

将文本以字符串数组的形式存储，每行一个字符串。然后使用parallel指令和schedule(static,1)子句将行分配给各个线程。

对行进行分词的一种方式是使用string.h头文件中的strtok函数。该函数的函数原型如下：

```
char* strtok(
      char*        string      /* in/out */,
      const char*  separators  /* in     */);
```

它在使用上有些不同寻常之处：当这个函数第一次被调用时，string参数是要被分析的文本，在我们的例子中它是输入的行。在接下来的调用中，它应该是NULL，因为在第一次调用时，strtok函数缓存了指向string的指针，在接下来的调用中，它从缓存的副本中返回连续的词
256 组。分离词组的字母应当在separators中传递，所以传递"\t\n"作为separators的参数。

**程序5-6 第一个多线程分词器**

```
 1  void Tokenize(
 2        char*  lines[]       /* in/out */,
 3        int    line_count    /* in     */,
 4        int    thread_count  /* in     */) {
 5     int my_rank, i, j;
 6     char *my_token;
 7
 8  #  pragma omp parallel num_threads(thread_count) \
 9        default(none) private(my_rank, i, j, my_token) \
10        shared(lines, line_count)
11     {
12        my_rank = omp_get_thread_num();
13  #     pragma omp for schedule(static, 1)
14        for (i = 0; i < line_count; i++) {
```

⊖ 这部分内容在第4章已经介绍过。如果你已经读过，可以跳过这部分内容。

```
15            printf("Thread %d > line %d = %s", my_rank, i,
                    lines[i]);
16            j = 0;
17            my_token = strtok(lines[i], " \t\n");
18            while ( my_token != NULL ) {
19               printf("Thread %d > token %d = %s\n", my_rank, j,
                       my_token);
20               my_token = strtok(NULL, " \t\n");
21               j++;
22            }
23         } /* for i */
24      }  /* omp parallel */
25
26   }  /* Tokenize */
```

基于这些假定，我们可以按照程序 5-6 来实现 Tokenize 函数。主函数已经初始化了 lines 数组，所以在 lines 中包含了输入文本，而 line_count 是存储在 lines 中字符串的数目。尽管从程序的功能来看，只需要 lines 作为输入参数，但是 strtok 函数对输入参数 lines 进行了修改。因此，当 Tokenize 函数返回时，lines 已经被修改过了。当以单线程的形式运行程序时，程序能正确地对输入流进行分词。程序的输入为

```
Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.
```

使用双线程第一次运行程序，得到了正确的结果。然而，再次运行程序时，得到下面的输出：

```
Thread 0 > line 0 = Pease porridge hot.
Thread 1 > line 1 = Pease porridge cold.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 1 > token 1 = cold.
Thread 0 > line 2 = Pease porridge in the pot
Thread 1 > line 3 = Nine days old.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Nine
Thread 0 > token 1 = days
Thread 1 > token 1 = old.
```

257

为什么会出现这种错误？回想一下，strtok 函数通过声明一个 static 存储类型的变量来缓存输入行，这将导致存储在这个变量中的值会从上一个调用保留到下一个调用。不幸的是，这个缓存的字符串变量是共享的，而不是私有的，因此线程 1 用第 2 行作为输入参数来对 strtok 进行调用，显然覆盖了线程 0 用第 1 行作为输入参数调用函数时变量中的值。更糟糕的是，线程 0 发现了一个单词（“days”），而这个单词本应该是线程 1 的输出。

因此 strtok 函数不是线程安全的：如果多个线程同时调用它，它可能产生不正确的输出。遗憾的是，C 语言库中的函数普遍不是线程安全的。例如头文件 stdlib.h 中的随机数产生器 random 和头文件 time.h 中的时间转换函数 localtime 都不是线程安全的。在某些情况下，C 标准提供了可选的、线程安全的函数版本。实际上，strtok 函数有一个线程安全的版本：

```
strtok:

    char* strtok_r(
          char*       string      /* in/out */,
          const char* separators  /* in     */,
          char**      saveptr_p   /* in/out */);
```

“_r”表明函数是可重入的，在某些情况下可重入与线程安全的意思是一样的。前两个参数

和 strtok 函数中的参数的作用相同，saveptr_p 参数被 strtok_r 函数用来跟踪函数正在处理字符串的哪个位置，它相当于一个在 strtok 函数中被缓存的指针。可以通过用 strtok_r 函数代替 strtok 函数来修正最初的 Tokenize 函数。我们仅需要声明一个 char * 变量作为第 3 个参数传递给函数，并分别用下面的代码替换第 17 行和第 20 行的调用

```
my_token = strtok_r(lines[i], " \t\n", &saveptr);
. . .
my_token = strtok_r(NULL, " \t\n", &saveptr);
```

### 不正确的程序可能会产生正确的结果

258 我们最初版本的分词程序有一个不易觉察的程序错误：第一次使用 2 个线程运行程序时，程序给出了正确的输出结果，第 2 次运行时，程序才给出了错误的结果。不幸的是，这种错误在并行程序中经常发生，而在共享内存程序中这种错误更加普遍。就像我们在本章开头所注意到的那样，大部分情况下，线程之间是相互独立运行的，语句执行序列具有不确定性。例如，我们不能假定线程 1 会首先调用 strtok 函数。如果线程 1 对 strtok 的调用发生在线程 0 已经对第 1 行分词完之后，那么对第 1 行进行分词的结果是正确的。然而，如果线程 1 对 strtok 函数的调用发生在线程 0 对第 1 行分词完成之前，那么很有可能线程 0 不能分析出第 1 行中所有的单词。所以在开发共享内存程序时，不要因为程序给出了正确的结果就认定程序是正确的。我们要小心对待线程之间的竞争条件。

## 5.11 小结

OpenMP 是一个共享内存系统上的编程标准，它使用专门的函数和预处理器指令 pragmas，所以与 Pthreads 和 MPI 不同，OpenMP 需要编译器的支持。OpenMP 最重要的特色之一就是它的设计使程序员可以逐步并行化已有的串行程序，而不是从零开始编写并行程序。

OpenMP 程序使用**多线程**而不是多进程。线程比进程更加轻量级，除了拥有自己的栈和程序计数器外，同一个进程的线程可以共享该进程几乎所有的资源。

为了使用 OpenMP 中的函数和宏，需要将 omp.h 头文件包含在 OpenMP 程序中。有几个 OpenMP 指令可以开启多线程，最常用的是 parallel 指令：

```
# pragma omp parallel
  structured block
```

这个指令告诉运行时系统并行执行下面的结构化块，即**派生**（fork）或者启动多个线程来执行该**结构化块**。结构化代码块是只有一个入口和一个出口的代码块，尽管结构化代码块允许调用 C 库函数 exit。被启动的线程的数目依赖于系统，但是大多数系统会为每一个内核开启一个线程。执行 block of code 的线程的集合叫做**线程组**。组中有一个线程在 parallel 指令之前执行，这个线程叫**主线程**，其余被 parallel 指令开启的线程叫做**从线程**。当所有的线程都结束后，从线程被终止或者**合并**（join），而主线程继续执行结构化代码块之后的代码。

许多 OpenMP 指令可以被**子句**（clauses）修改。使用最频繁的是 num_threads 子句，当使
259 用 OpenMP 指令启动线程组时，可以通过修改 num_threads 子句来启动需要数目的线程。

当 OpenMP 启动了一组线程后，给每一个线程分配一个线程编号（rank），编号的范围是 0，1，…，thread_count - 1。调用 OpenMP 库函数 omp_get_thread_num 返回调用该函数的线程编号。函数 omp_get_threads 返回当前线程组中线程的数目。

开发共享内存程序时的一个主要问题是可能存在**竞争条件**。竞争条件发生在多个线程都试图访问一个共享资源，且至少有一个访问是更新的情况下。这些访问可能会产生错误。每次只能被

一个线程更新的共享变量的代码叫做**临界区**。因此，如果多个线程试图更新一个共享变量，程序就会产生竞争条件，更新该变量的代码就成为临界区。OpenMP 提供了多种机制实现对临界区的**互斥**访问。我们学习了其中的的四种：

1）Critical 指令确保一次只有一个线程执行结构化代码块。如果多个线程试图执行临界区中的代码，除了其中的某个线程外，其他所有的线程都将在临界区前被阻塞。当该线程离开临界区后，其他线程才会获得进入临界区的机会。

2）程序中可以使用命名的 critical 指令，名字不同的临界区能被同时执行。多个线程在处理多个同名临界区时会按照处理未命名临界区的方式处理，但是多个线程可以同时进入有着不同名字的临界区。

3）atomic 指令只能用在形式为 x < op > = < expression > 、x + +、+ + x、x - -，或者 - - x的临界区中。这个指令可以利用特殊的硬件指令来实现，所以它比普通的临界区执行速度快。

4）简单锁是最通用的互斥方式，它使用函数调用实现对临界区的互斥访问：

```
omp_set_lock(&lock);
critical section
omp_unset_lock(&lock);
```

当多个线程调用 omp_set_lock 时，只有一个线程可以进入临界区，其他线程将被阻塞。直到第一个线程调用 omp_unset_lock 后，其他线程才可能获得机会进入临界区。

所有的互斥机制都会导致死锁这样的严重问题，所以对它们的使用需要十分小心。

for 指令可以将 for 循环中的迭代在线程间进行划分。这个指令不是开启一组线程，而是将 for 循环中的迭代划分给已经存在的线程组中的线程。如果在划分的同时想要开启一组线程，则可以使用 parallel for 指令。对于能够并行化的 for 循环的形式有一些限制：最基本的是在循 260
环开始执行前，运行时系统必须能够确定循环体迭代的次数。详见程序 5-3。

然而上面的限制还不足以确保 for 循环满足规定的形式，它还必须没有任何形式的**循环依赖**。循环依赖发生在一个内存位置在一次迭代中被读或写后，又在另一次迭代中被写。OpenMP 不会探测循环依赖，发现和消除循环依赖的工作由程序员负责。然而，有时循环依赖不能够被消除，此时该循环不能被并行化。

在缺省情况下，绝大多数系统在并行化的 for 循环中对迭代使用**块划分**。如果循环总共有 $n$ 次迭代，那么一般将最初的 $n$/thread_count 次迭代分配给线程 0，接下来的 $n$/thread_count 次迭代分配给线程 1，以此类推。然而 OpenMP 提供了许多**调度**选项。调度子句有着下面的形式：

```
schedule(<type> [,<chunksize>])
```

type 可以是 static、dynamic、guided、auto 或者 runtime。在 static 调度中，迭代在循环开始执行前分配给线程。在 dynamic 和 guided 调度中，迭代在执行过程中分配给线程。当线程完成一个迭代块（一块连续的迭代）后，它请求另外一个迭代块。如果采用 auto 调度，调度策略由编译器或者运行时系统决定。如果采用 runtime 调度，那么调度策略将在运行时通过检查环境变量 OMP_SCHEDULE 的值来决定。

只有 static、dynamic 和 guided 调度有 chunksize。在 static 调度中，chunksize 大小的迭代块以轮转的方式分配给线程。在 dynamic 调度中，每个线程分配 chunksize 个迭代，当某个线程完成它的迭代块后，它会请求另一个迭代块。在 guided 调度中，迭代块的大小随着迭代的进行而减小。

在 OpenMP 中，变量的**作用域**是可以访问该变量的线程的集合。通常在 OpenMP 指令前定义

的变量在内部构造中有**共享**作用域，也就是说，所有的线程都能访问该变量。上述情况不适用于定义在 for 循环或者 parallel for 构造中的变量，它们是**私有的**，也就是说，每个线程都有该变量的副本。定义在 OpenMP 构造中的变量有私有作用域，因为它们分配给在正在执行的线程的栈中。

作为一个经验法则，显式地赋予变量作用域是个很好的主意。这可以通过使用作用域子句

```
default(none)
```

修改 parallel 或者 parallel for 指令来实现。这条子句告诉系统每一个在 OpenMP 构造中使用的变量的作用域必须被显式定义。在大多数情况下，变量作用域的定义可以通过 private 或
261 者 shared 子句实现。

我们遇到的唯一例外是**归约变量**。**归约操作符**是一个二元操作符（例如，加或者乘），而一次归约计算对一个操作数序列重复使用相同的归约操作符，从而得到一个唯一的结果。此外，所有操作的中间结果应该存储在同一个变量——**归约变量**中。例如，假设 $A$ 是有 $n$ 个元素的数组，那么下面的代码

```
int sum = 0;
for (i = 0; i < n; i++)
   sum += A[i];
```

是一个归约操作，归约操作符是加法，归约变量是 sum。如果我们试图并行化这个循环，归约变量应该同时具有私有变量和共享变量的性质。开始，我们希望每个线程将数组元素加到它私有的 sum 中，但是在线程结束时，我们希望私有的 sum 结合到一个单独的、共享的 sum 中。因此 OpenMP 提供了归约子句来识别归约变量和操作符。

barrier 指令可以阻塞同一组中的线程直到所有的线程都到达该指令。我们已经看到 parallel、parallel for 和 for 指令在结构化代码块的末尾都有隐式的路障。

现代的微处理器架构使用缓存以减少主存访问时间，所以典型的体系结构都有专门的硬件确保在不同处理器芯片上的缓存是**一致的**。因为缓存一致性的基本单位，**缓存行**或**缓存块**，一般比一个主存字大，所以可能产生一些副作用：两个线程可能访问内存中的不同位置，但是当这两个位置属于同一个缓存行时，缓存一致性硬件所表现出来的处理方式就好像这两个线程访问的是内存中的同一个位置——如果其中一个线程更新了它所访问的主存地址的值，那么另外一个变量试图读取它要访问的主存地址时，它不得不从主存获取该值。也就是说，硬件强制该线程表现得好像它共享了变量，因此这种情况称为**伪共享**，这会大大降低共享内存程序的性能。

某些 C 函数通过声明 static 变量，在不同调用之间缓存数据。当多个线程调用该函数时，这可能导致错误。因为静态存储在多个线程间共享，一个线程可以写覆盖另外一个线程的数据。这样的函数不是**线程安全的**。不幸的是，在 C 函数库中有一些这样的函数，然而有时非线程安全的函数在库中可以找到其线程安全的版本。

在我们的程序中，我们看到了一个很不容易觉察的问题：当我们固定程序的输入，并用多个线程运行程序时，尽管程序有错，但它有时还是会给出正确的结果。程序在测试期间给出正确的
262 结果并不能保证程序是正确的，需要我们自己去发现可能的竞争条件。

## 5.12 习题

5.1 如果已经定义了宏_OPENMP，它是一个 int 类型的十进制数。编写一个程序打印它的值。这个值的意义是什么？

5.2 从本书的网站上下载 omp_trap_1.c，并且删除 critical 指令。用越来越多的线程和越来越大的 $n$ 来编译和运行程序。当程序第一次出现错误时，线程和梯形的数目分别是多少？

5.3　按照下面的要求修改 omp_trap_1.c

a. 它使用代码块

b. 通过使用 OpenMP 函数 omp_get_wtime()对 parallel 代码块计时。语法是

```
double omp_get_wtime(void)
```

它返回从过去的某个时间开始，已经过去多少秒。计时的细节请见 2.6.4 节。另外，需要注意的是 OpenMP 有 barrier 指令：

```
#  pragma omp barrier
```

现在找出一个至少有两个核的系统，统计程序在如下情况下的运行时间，

c. 一个线程和一个值较大的 $n$

d. 两个线程和同样大小的 $n$

分别会产生什么样的结果？从本书的网站上下载 omp_trap_2.c。该程序的性能与 omp_trap_1.c 的性能相比如何？解释你的答案。

5.4　OpenMP 为归约变量创建私有变量，这些私有变量的值按照归约操作符的类型初始化。例如，如果归约操作符是加法，那么私有变量初始化为 0；如果归约操作符是乘法，那么私有变量初始化为 1。当操作符分别为 &&、||、&、|、^时，私有变量初始化为什么？

5.5　假定在 Bleeblon 计算机上，浮点型变量能够存储小数点后 3 位数字，它的浮点寄存器可以存储小数点后 4 位，并且在任意的浮点操作后，结果在存储前被四舍五入为小数点后 3 位。现在假设一个 C 程序声明了一个数组：

```
float a[] = {4.0, 3.0, 3.0, 1000.0};
```

a. 如果在 Bleeblon 计算机上运行下面的代码块，输出会是什么？

```
int i;
float sum = 0.0;
    for (i = 0; i < 4; i++)
       sum += a[i];
    printf("sum = %4.1f\n", sum);
```

263

b. 考虑如下的代码

```
   int i;
   float sum = 0.0;
#  pragma omp parallel for num_threads(2) \
      reduction(+:sum)
   for (i = 0; i < 4; i++)
      sum += a[i];
   printf("sum = %4.1f\n", sum);
```

假设运行时系统将迭代 i = 0，1 分配给线程 0，将迭代 i = 2，3 分配给线程 1，那么在 Bleeblon 计算机上，该程序的输出是什么？

5.6　编写一个 OpenMP 程序，确定并行 for 循环的默认调度方式。程序的输入应该是迭代的次数，而程序的输出是循环中的每次迭代被哪一个线程执行。例如，现在有两个线程和四次迭代，那么输出可能是：

```
Thread 0: Iterations 0 — 1
Thread 1: Iterations 2 — 3
```

5.7　我们第一次试图并行化 π 值估计程序其实是不正确的。实际上，我们以程序在单线程条件下运行的结果作为依据，证明了程序在双线程下运行时给出了错误的结果。请解释我们为什么“信任”程序在单线程下运行所得到的结果。

5.8　考虑下面的循环

```
a[0] = 0;
for (i = 1; i < n; i++)
   a[i] = a[i-1] + i;
```

在这个循环中显然有循环依赖，因为在计算 a[i] 前必须先算 a[i-1] 的值。请你找到一种方法消除循环依赖，并且并行化这个循环。

5.9 使用 parallel for 指令来修改梯形积分法程序（omp_trap_3.c），使之可以使用 schedule (runtime)子句来修改 parallel for 指令。给环境变量 OMP_SCHEDULE 赋予不同的值，然后运行该程序，并确定每次迭代由哪个线程执行。上面的工作可以通过声明一个有 $n$ 个 int 类型变量的数组 iterations，并在 Trap 函数中将循环第 $i$ 次迭代对函数 omp_get_thread_num()调用的返回结果赋给 iterations[i]得以实现。你的系统默认的循环分配方式是什么？guided 调度又是怎么被确
264 定的？

5.10 被未命名的 critical 指令修改的结构化代码块构成了一个单独的临界区。那么，当有多条 atomic 指令，这些指令修改的是不同的变量时，它们是不是都被视为各自独立的临界区？

可以编写一个小程序来回答上面的问题。基本思想是让所有的线程同时执行类似下面的代码：

```
   int i;
   double my_sum = 0.0;
   for (i = 0; i < n; i++)
#     pragma omp atomic
      my_sum += sin(i);
```

可以用 parallel 指令修改上面的代码：

```
#  pragma omp parallel num_threads(thread_count)
   {
      int i;
      double my_sum = 0.0;
      for (i = 0; i < n; i++)
#        pragma omp atomic
         my_sum += sin(i);
   }
```

由于 my_sum 和 i 都是在 parallel 块中声明的，所以每个线程自己都有这两个变量的私有副本。现在如果我们统计在较大的 $n$ 的情况下，程序对于 thread_ count = 1 和 thread_ count > 1 的运行时间，就会发现只要 thread_ count 小于系统可用的核的数量。如果不同线程对 my_ sum + = sin(i)的执行被视为不同的临界区，那么在单线程和多线程环境下，程序的运行时间大致相同。如果不同线程对 my_ sum + = sin(i)的执行被视为相同的临界区，那么多线程环境下的运行时间应当比单线程环境下程序的运行时间长很多。编写一个 OpenMP 程序实现上述测试，然后判断在更新操作被 atomic 指令保护时，你所使用的 OpenMP 实现是否允许同时执行更新操作？

5.11 在 C 语言中，以二维数组作为参数的函数必须在参数列表中说明列数，所以 C 程序员通常只使用一维数组，然后用代码中显示地将二维下标转换为一维下标。修改本书中的 OpenMP 矩阵 - 向量乘法程序，采用一维数组表示矩阵。

5.12 从本书的网站上下载源代码 omp_mat_vect_rand_split. c。寻找一个工具（例如 valgrind [49]）进行缓存分析，并且按照缓存分析器的文档来编译程序（例如，在使用 valgrind 时，需要符号表和完
265 全优化，使用命令 gcc -g -O2 …）。现在按照缓存分析器文档中的指令来运行程序，程序的输入分别为 $k \times (k \cdot 10^6)$、$(k \cdot 10^3) \times (k \cdot 10^3)$ 和 $(k \cdot 10^6) \times k$。$k$ 的值应该足够大，使得上面的 3 个输入中至少有一个可以使得 2 级缓存的失效次数的数量级为 $10^6$。

a. 每种输入各有多少次 1 级缓存写缺失？

b. 每种输入各有多少次 2 级缓存写缺失？

c. 大部分的写缺失在哪里发生？哪一种输入写缺失最多？请解释为什么。

d. 每种输入各有多少次 1 级缓存读缺失？

e. 每种输入各有多少次 2 级缓存读缺失？

f. 大部分的读缺失在哪里发生？哪一种输入读缺失最多？请解释为什么。

g. 用上面的 3 个输入分别运行程序，但是不再使用缓存分析器。在哪一种输入下，程序运行最快？在哪一种输入下，程序运行最慢？请结合缓存缺失来解释你观察到的不同。

5.13 我们考察 8000 × 8000 作为之前的矩阵 - 向量乘法程序的输入时该程序的性能。假定线程 0 和线程 2 被分配给了不同的处理器。如果一个缓存行包含 64 字节或者 8 个双精度数，那么在线程 0 和线程 2 之间的伪共享可不可能发生在向量 y 的任何地方？为什么？如果线程 0 和线程 3 被分配给了不同的处理器，那么伪共享可不可能发生在向量 y 的任何地方？

5.14 我们考察 8 × 8 000 000 作为之前的矩阵 - 向量乘法程序的输入时该程序的性能。假定双精度数大小是 8 字节，而缓存行的大小是 64 字节，且系统有两个双核处理器。

a. 为了存储向量 y，最少需要多少个缓存行？

b. 为了存储向量 y，最多需要多少个缓存行？

c. 如果缓存行的边界和双精度数的边界始终一致，那么一共有多少种方式将 y 中的元素分配给缓存行？

d. 如果我们只考虑两个线程共享一个处理器，那么在我们的计算机上一共有多少种方式将四个线程分配给处理器？我们假定在同一个处理器上的内核共享缓存。

e. 在我们的例子中，有没有哪一种对向量的分配和对线程的分配方式不会产生伪共享？换句话说，对于处于不同处理器上的线程，它们各自要处理的向量的元素不会出现在同一个缓存行内。

f. 有多少种方法将向量元素分配给缓存行和将线程分配给处理器？

g. 在这些分配中，有多少会使得程序没有伪共享？ 266

5.15 a. 修改矩阵 - 向量乘法程序，使得可能产生伪共享时，程序会填充向量 y。当线程以锁步的方式执行时，填充应当确保在一个缓存行内的 y 的元素不会被两个或者两个以上的线程共享。例如，假设一个缓存行包含 8 个双精度数，我们用 4 个线程来运行程序。如果我们为 y 至少分配了 48 个双精度数的存储空间，那么在每次 `for i` 循环的迭代中，不可能会有两个线程同时访问同一个缓存行。

b. 修改矩阵 - 向量乘法程序，使得每个线程在 `for i` 循环时对它要处理的向量 y 的元素采用私有存储。当一个线程计算出它私有的 y 中的元素后，它应该将这些私有的变量复制到共享变量中。

c. 以上两种方法与最初的方法相比，性能如何？两者相比，性能如何？

5.16 尽管 `strtok_r` 函数是线程安全的，但是它会不必要地修改输入字符串。请编写一个线程安全的分词函数，该函数不会修改输入字符串。

## 5.13 编程作业

5.1 使用 OpenMP 实现第 2 章讨论的并行直方图绘制程序。

5.2 假定我们往一个正方形靶子上随机投掷飞镖，靶心位于正中，靶子的长度为 2 英尺。假设有一个圆内切于该正方形靶子，那么圆的半径是 1 英尺，面积为 π 平方英尺。如果飞镖击中的点是均匀分布的（且飞镖总是击中靶子），那么飞镖落在圆内的次数应当近似地满足下面的等式：

$$\frac{\text{飞镖落在圆内的次数}}{\text{飞镖落在靶内的总次数}} = \frac{\pi}{4}$$

因为环所包含的面积与正方形面积的比值是 π/4。

我们可以用这个公式和随机数产生器来估计 π 的值：

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

这种采用了随机（随机投掷）的方法称为蒙特卡洛（Monte Carlo）方法。

编写一个采用蒙特卡洛方法的 OpenMP 程序估计 $\pi$ 的值。在开启任何线程前读取总的投掷次数。使用 reduction 子句计算飞镖击中环内的次数。在合并所有的线程后，打印结果。为了估计出一个合理的 $\pi$，投掷次数必须非常大，可能总的投掷次数和击中环内的次数都得用 long int 型来表示。

5.3 计数排序是一个简单的串行程序，实现如下：

```
void Count_sort(int a[], int n) {
   int i, j, count;
   int* temp = malloc(n*sizeof(int));

   for (i = 0; i < n; i++) {
      count = 0;
      for (j = 0; j < n; j++)
         if (a[j] < a[i])
            count++;
         else if (a[j] == a[i] && j < i)
            count++;
      temp[count] = a[i];
   }

   memcpy(a, temp, n*sizeof(int));
   free(temp);
} /* Count_sort */
```

267

基本思想是对列表 a 中的每一个元素 a[i]，找出比 a[i]小的元素的个数。然后用该结果作为下标，将 a[i]插入新的列表中。当列表中包含相同的元素时，计数排序有点小小的问题，因为此时它们会被放到新列表中的同一个位置。为了解决这个问题，可以递增相同元素的下标。如果 a[i] == a[j] 且 j < i，那么认为 a[j] “小于” a[i]。

在算法执行结束后，通过字符串库函数 memcpy 将临时列表中的元素复制到最初的列表中。

a. 如果我们试图并行化 for 循环（外层循环），哪些变量应当是私有的，哪些变量应当是共享的？

b. 如果我们使用前面定义的作用域来并行化 for 循环，是否存在循环依赖？请解释你的回答。

c. 我们是否能够并行化对函数 memcpy 的调用？我们是否能够修改代码，使得这部分代码可以被并行化？

d. 编写一个包含对计数排序程序并行化实现的 C 程序。

e. 与串行化的计数排序程序相比，并行化的计数排序程序的性能如何？与串行化的库函数 qsort 相比，并行化的计数排序程序的性能如何？

268

5.4 解线性方程组时，我们经常采用高斯消元法和回代法。高斯消元法通过行操作将一个 $n \times n$ 矩阵转换为上三角矩阵。

- 将一行加到另外一行上
- 两行互换
- 将一行乘以一个非 0 常数

在一个上三角矩阵中，左上角到右下角的对角线下的元素全为 0。

例如，线性方程组：

$$\begin{aligned} 2x_0 - 3x_1 &= 3 \\ 4x_0 - 5x_1 + x_2 &= 7 \\ 2x_0 - x_1 - 3x_2 &= 5 \end{aligned}$$

可以整理为上三角矩阵：

$$\begin{aligned} 2x_0 - 3x_1 &= 3 \\ x_1 + x_2 &= 1 \\ -5x_2 &= 0 \end{aligned},$$

这个上三角矩阵可以很容易地解出：先通过最后一个等式得到 $x_2$，然后通过第二个等式得到 $x_1$，最后

用第一个等式得到 $x_0$。

我们可以为代入法设计一些串行算法。其中“面向行”的版本是：

```
for (row = n-1; row >= 0; row--) {
   x[row] = b[row];
   for (col = row+1; col < n; col++)
      x[row] -= A[row][col]*x[col];
   x[row] /= A[row][row];
}
```

这里，线性方程组右侧的结果变量存储在向量 b 中，二维数组的系数存储在数组 A 中，而答案存储在数组 x 中。另外一种“面向列”的方法如下：

```
for (row = 0; row < n; row++)
   x[row] = b[row];

for (col = n-1; col >= 0; col--) {
   x[col] /= A[col][col];
   for (row = 0; row < col; row++)
      x[row] -= A[row][col]*x[col];
}
```

a. 判断“面向行”的算法的外层循环是否可并行化。 269

b. 判断“面向行”的算法的内层循环是否可并行化。

c. 判断“面向列”的算法的外层循环是否可并行化。

d. 判断“面向列”的算法的内层循环是否可并行化。

e. 为你认为可以并行化的循环语句编写一个 OpenMP 程序。single 指令在这里可能会有用——如果一个代码块被并行执行，子代码块可以被#pragma omp single 指令修改，那么它的子代码块应该只能被一个线程执行。正在执行的线程组中的线程会在这个指令的末尾被阻塞，直到所有的线程都完成。

f. 用 schedule(runtime)子句修改你的并行循环，并用多种调度方式测试你的程序。如果你的上三角矩阵有 10 000 个变量，哪一种调度策略性能最好？

5.5　使用 OpenMP 编写一个程序实现高斯消元法（参见前面的问题）。你可以假定输入的方程组不需要行交换。

5.6　使用 OpenMP 实现生产者 - 消费者程序，其中一些线程是生产者，另外一些线程是消费者。在文件集合中，每个生产者针对一个文件，从文件中读取文本。它们将读出的文本行插入到一个共享的队列中。消费者从队列中取出文本行，并对文本行进行分词。符号是被空白符分开的单词，当消费者发现一个单词后，它将该单词输出到 stdout。

第6章
An Introduction to Parallel Programming

# 并行程序开发

在前面的三章中，我们不仅学习了并行应用程序编程接口，也开发了一些小的并行程序，这些并行程序涉及并行算法的实现。在本章中，我们会看到一些更大型的例子：$n$ 体问题和旅行商问题。对于每一个问题，我们先提出串行算法，然后对串行算法做出改进。在运用 Foster 方法并行化程序时，我们发现开发共享内存程序和分布式内存程序有很多相同之处。我们还发现有些并行问题找不到相似的串行问题。作为并行程序员，我们发现有些问题必须从零开始，重新设计程序。

## 6.1 $n$ 体问题的两种解决方法

在 $n$ 体问题中，我们试图确定一些相互作用的粒子在一段时间后的位置和速度。例如，天文学家可能想知道某些星球的位置和速度，而化学家可能想知道分子或者原子的位置和速度。$n$ 体问题的解决方案是通过模拟粒子行为找到解决 $n$ 体问题的程序。问题的输入是粒子的质量，粒子在开始时的位置和速度；输出一般是粒子在用户指定的时间序列中的速度和位置，或者只是粒子在用户指定的时间段后的速度和位置。

我们首先开发一个串行化的 $n$ 体程序，然后针对共享内存和分布式内存系统并行化该串行程序。

### 6.1.1 问题

为了使问题更加清楚，我们的程序模拟星球或者行星的运行。我们使用牛顿第二运动定律和万有引力定律来确定位置和速度。因此，如果粒子 $q$ 在时间 $t$ 时的位置为 $\boldsymbol{s}_q(t)$，粒子 $k$ 在时间 $t$
271 时的位置为 $\boldsymbol{s}_k(t)$，那么粒子 $k$ 作用在粒子 $q$ 上的作用力为

$$\boldsymbol{f}_{qk}(t) = -\frac{Gm_q m_k}{|\boldsymbol{s}_q(t) - \boldsymbol{s}_k(t)|^3}[\boldsymbol{s}_q(t) - \boldsymbol{s}_k(t)] \tag{6-1}$$

$G$ 是万有引力常数（$6.673 \times 10^{-11}\ \mathrm{m}^3/(\mathrm{kg} \cdot \mathrm{s}^2)$），$m_q$ 和 $m_k$ 分别是粒子 $q$ 和 $k$ 的质量，$|\boldsymbol{s}_q(t) - \boldsymbol{s}_k(t)|$ 是粒子 $q$ 和 $k$ 之间的距离。需要注意的是，位置、速度、加速度和力一般都是向量，所以用黑斜体表示这些变量，用斜体表示其他标量、变量，例如时间 $t$ 和万有引力常数 $G$。

对于任意一个粒子，使用式（6-1），通过累加所有其他粒子在该粒子上的作用力，从而得到该粒子所受到的总作用力。假设粒子的编号为 0、1、2、…、$n-1$，那么粒子 $q$ 所受到的总作用力是：

$$\boldsymbol{F}_q(t) = \sum_{\substack{k=0\\k\neq q}}^{n-1} \boldsymbol{f}_{qk} = -Gm_q \sum_{\substack{k=0\\k\neq q}}^{n-1} \frac{m_k}{|\boldsymbol{s}_q(t) - \boldsymbol{s}_k(t)|^3}[\boldsymbol{s}_q(t) - \boldsymbol{s}_k(t)] \tag{6-2}$$

需要注意的是，加速度是位移的二阶导数，牛顿第二运动定律表明作用在物体上的作用力等于物体的质量乘以它的加速度，所以如果粒子 $q$ 的加速度是 $\mathbf{a}_q(t)$，那么 $\boldsymbol{F}_q(t) = m_q\mathbf{a}_q(t) = m_q\boldsymbol{s}_q''(t)$，其中 $\boldsymbol{s}_q''(t)$ 是位移 $\boldsymbol{s}_q(t)$ 的二阶导数。因此我们可以使用式（6-2）得到粒子 $q$ 的加速度：

$$\boldsymbol{s}_q''(t) = -G\sum_{\substack{j=0\\j\neq q}}^{n-1} \frac{m_j}{|\boldsymbol{s}_q(t) - \boldsymbol{s}_j(t)|^3}[\boldsymbol{s}_q(t) - \boldsymbol{s}_j(t)] \tag{6-3}$$

牛顿定律给了我们一个微分方程（涉及导数的方程），我们的工作是计算出时间 $t$ 时粒子的位移 $s_q(t)$ 和速度 $v_q(t)=s'_q(t)$。

假定需要计算出在时间序列

$$t=0,\ \Delta t,\ 2\Delta t,\ \cdots,\ T\Delta t$$

下粒子的位移和速度，更为常见的情况是，我们仅仅需要计算出 $T\Delta t$ 时粒子的位移和速度。$T$ 和 $\Delta t$ 由用户说明，所以程序的输入是 $n$（即粒子的个数）、$\Delta t$、$T$ 和每个粒子的质量、初始位置和初始速度。在通用的解决方案中，位移和速度都是三维向量，但是为了使问题更加简化，假定粒子在平面上移动，使用二维向量来表示它们。

程序的输出是 $n$ 个粒子在时间步长 0，$\Delta t$，$2\Delta t$，…，$T\Delta t$ 时的位置和速度，或者只是粒子在时间 $T\Delta t$ 时的位置和速度。为了得到在最终时间的输出，可以增加一个输入选项，用户可以通过这个选项说明只需要最终的位置和速度。 272

### 6.1.2 两个串行程序

串行的 $n$ 体解决方法一般以下面的伪代码为基础：

```
1     Get input data;
2     for each timestep {
3        if (timestep output) Print positions and velocities of
             particles;
4        for each particle q
5           Compute total force on q;
6        for each particle q
7           Compute position and velocity of q;
8     }
9     Print positions and velocities of particles;
```

可以使用计算粒子总作用力的公式（式（6-2））来细化上面伪代码中计算作用力的第 4～5 行：

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

这里，假设粒子所受到的作用力 forces 和位置 pos 以二维数组的形式存储。我们定义常量 X = 0 和 Y = 1，所以粒子在 $x$ 轴上所受的作用力为 forces[q][X]，在 $y$ 轴上受到的作用力为 forces[q][Y]（稍后我们将进一步研究程序的数据结构）。

按照牛顿第三运动定律，作用力和反作用力是成对出现的。利用牛顿第三运动定律将作用力的计算减半，如果粒子 $k$ 对粒子 $q$ 的作用力是 $\boldsymbol{f}_{qk}$，那么 $q$ 对 $k$ 的作用力是 $-\boldsymbol{f}_{qk}$。通过简化，我们可以按照程序 6-1 修改计算作用力的代码。为了更好地理解这个伪代码，我们将作用力设想为一个二维数组：

$$\begin{bmatrix} 0 & \boldsymbol{f}_{01} & \boldsymbol{f}_{02} & \cdots & \boldsymbol{f}_{0,n-1} \\ -\boldsymbol{f}_{01} & 0 & \boldsymbol{f}_{12} & \cdots & \boldsymbol{f}_{1,n-1} \\ -\boldsymbol{f}_{02} & -\boldsymbol{f}_{12} & 0 & \cdots & \boldsymbol{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\boldsymbol{f}_{0,n-1} & -\boldsymbol{f}_{1,n-1} & -\boldsymbol{f}_{2,n-1} & \cdots & 0 \end{bmatrix}$$

（为什么所有对角线上的元素都是0?）最初的算法简单地将第 $q$ 行的所有元素相加得到 forces
273 [q]。在修改过的算法中，当 $q=0$ 时，循环体 for each particle q 将所有第0行的元素加到 forces[0]中。对 $k=1, 2, \cdots, n-1$，第0列的第 $k$ 个元素会加到 forces[k]中。通常，第 $q$ 次迭代将第 $q$ 行在对角线右边的元素加到 forces[q]中，然后将第 $q$ 列在对角线以下的元素分别加到相应粒子的作用力中，也就是说，将第 $k$ 个元素加到 forces[k]中。

**程序 6-1　计算 $n$ 体作用力的简化算法**

```
for each particle q
   forces[q] = 0;
for each particle q {
   for each particle k > q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
      force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff

      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
}
```

使用修改过的算法时，必须在另外一个循环中初始化 forces 数组，因为计算作用力的第 $q$ 次迭代也会向 forces[k] ($k=q+1, q+2, \cdots, n-1$) 加入值，而不只是计算 forces[q]。

为了区分这两种算法，我们把最初计算作用力的 $n$ 体算法叫做基本算法，改进后的算法叫做简化算法。

现在还需要计算出位置和速度。我们知道粒子 $q$ 的加速度可以通过公式

$$\boldsymbol{a}_q(t) = \boldsymbol{s}_q''(t) = \boldsymbol{F}_q(t)/m_q$$

计算得出，$\boldsymbol{s}_q''(t)$ 是 $\boldsymbol{s}_q(t)$ 的二阶导数，$\boldsymbol{F}_q(t)$ 是粒子 $q$ 在时间 $t$ 时所受到的作用力。我们也知道速度 $\boldsymbol{v}_q(t)$ 是位移 $\boldsymbol{s}_q'(t)$ 的一阶导数，所以我们需要对加速度求积分得到速度，然后通过对速度求积分得到位移。

开始时，我们认为可以很容易地找到在式（6-3）中函数的不定积分。然而，仔细思考以后
274 我们发现这种方法有问题：等式右边包含未知的函数 $\boldsymbol{s}_q$ 和 $\boldsymbol{s}_k$（不仅仅是时间 $t$），所以我们采用数值分析方法来估计位移和速度。这表示，我们不是找到一个简单的闭公式，而是逼近感兴趣的时间点上的位移和速度。有许多可以采用的数值分析方法，但是我们采用了最简单的一种：欧拉方法，这个方法以著名瑞士数学家欧拉（1707—1783年）命名。在欧拉方法中，使用切线逼近一个函数。它的基本思想是：如果我们在时间 $t_0$ 知道函数 $g(t_0)$ 的值以及它的导数 $g'(t_0)$，那么我们在时间 $t_0+\Delta t$ 时通过 $g(t_0)$ 处的切线近似地得到 $g(t_0+\Delta t)$。图6-1是该方法的一个例子。现在如果我们知道了线上的一个点（$t_0$, $g(t_0)$），和线的斜率 $g'(t_0)$ 那么直线的方程可以表示为

$$y = g(t_0) + g'(t_0)(t - t_0)$$

因为我们只感兴趣在 $t=t_0+\Delta t$ 时函数的值，所以有

$$g(t+\Delta t) \approx g(t_0) + g'(t_0)(t+\Delta t - t) = g(t_0) + \Delta t g'(t_0)$$

注意，当 $g(t)$ 和 $y$ 是向量时，上面的公式仍然起作用：当 $g(t)$ 和 $y$ 是向量时，$g'(t)$ 也是向量，这个公式只是将向量与另一个标量 $\Delta t$ 相乘的向量相加。

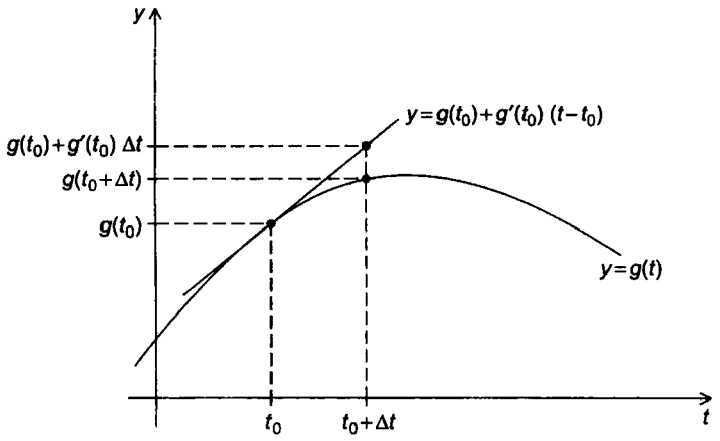



图 6-1　使用切线来逼近一个函数

现在我们知道在时间 0 时的 $s(t)$ 和 $s'(t)$，我们要使用切线以及计算加速度的公式计算 $s_q(\Delta t)$ 和 $v_q(\Delta t)$：

$$s_q(\Delta t) \approx s_q(0) + \Delta t s'_q(0) = s_q(0) + \Delta t v_q(0)$$

$$v_q(\Delta t) \approx v_q(0) + \Delta t v'_q(0) = v_q(0) + \Delta t a_q(0) = v_q(0) + \Delta t \frac{1}{m_q} F_q(0)$$

当我们想扩展这个方法来计算 $s_q(2\Delta t)$ 和 $s'_q(2\Delta t)$，我们看到事情变得有些不同，因为我们不知 275
道 $s_q(\Delta t)$ 和 $s'_q(\Delta t)$ 的准确值。然而，如果对 $s_q(\Delta t)$ 和 $s'_q(\Delta t)$ 求出的近似值是比较好的，那么应该可以通过相同的方法计算得到 $s_q(2\Delta t)$ 和 $s'_q(2\Delta t)$ 比较好的逼近。这就是欧拉公式所要做的（见图 6-2）。现在我们可以加入计算位置和速度的代码以完善两种 $n$ 体问题算法的伪代码：

```
pos[q][X] += delta_t*vel[q][X];
pos[q][Y] += delta_t*vel[q][Y];
vel[q][X] += delta_t/masses[q]*forces[q][X];
vel[q][Y] += delta_t/masses[q]*forces[q][Y];
```

这里，分别用 pos[q]、vel[q]和 forces[q]存储粒子 $q$ 的位置、速度和所受到的作用力。

在并行化该串行程序之前，我们先研究一下算法的数据结构。我们用数组类型来存储向量：

```
#define DIM 2

typedef double vect_t[DIM];
```

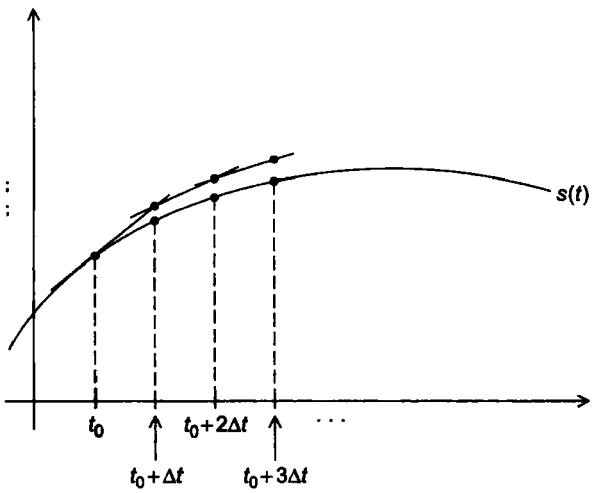



图 6-2　欧拉方法

也可以使用结构体代替数组存储向量。然而，如果使用数组，当程序处理三维问题时，原则上，只需要改变宏 DIM。如果使用结构体，那么必须重写代码以访问向量中的元素。

对于每一个粒子，需要知道它的：

- 质量。
- 位置。 276
- 速度。

- 加速度。
- 所受作用力。

由于我们采用的是牛顿物理学，所以每一个粒子的质量都是常数，但是其他的值会随着程序的运行而不断改变。如果我们查看代码会发现，每当计算出这些变量在某个时间点的新值时，它原来的值就不再需要了。例如，我们不会进行如下的操作：

```
new_pos_q = f(old_pos_q);
new_vel_q = g(old_pos_q, new_pos_q);
```

此外，加速度只用来计算速度，它的值可以通过计算作用力得出，所以我们只需要用一个临时的本地变量存储加速度。

对于每一个粒子，我们只需要存储它的质量、当前位置、速度和所受作用力。可以用结构体来存储这四个变量，用结构体数组存储所有粒子的数据。当然，我们不是非得将与粒子相关的变量存储在结构体中，也可以用多种方式将数据分离在不同的数组中。我们选择将质量、位置和速度存储在一个结构体中，而将粒子所受作用力存储在数组中。这样作用力会存储在主存中连续的地址中，我们可以使用快速的函数，例如 memset，在迭代开始的时候将所有的元素置为0：

```
#include <string.h>  /* For memset */
. . .
vect_t* forces = malloc(n*sizeof(vect_t));
. . .
for (step = 1; step <= n_steps; step++) {
   . . .
   /* Assign 0 to each element of the forces array */
   forces = memset(forces, 0, n*sizeof(vect_t);
   for (part = 0; part < n-1; part++)
      Compute_force(part, forces, . . .)
   . . .
}
```

如果粒子所受的作用力是结构体的成员，那么它不会在主存中占据连续的区域，这样我们必须使用较慢的 for 循环将每个元素置为0。

### 6.1.3 并行化 n 体算法

对 $n$ 体算法使用 Foster 方法。我们最初的目的是要运行多个任务，开始时可以将每一个时间点上对位置、速度和所受作用力的计算视为一个任务。在基本算法中，粒子所受的总作用力可以
277 直接通过作用力式（6-2）计算得出。粒子 $q$ 在时间 $t$ 时所受作用力 $\boldsymbol{F}_q(t)$ 的计算需要每个粒子 $r$ 的位置 $\boldsymbol{s}_r(t)$。对速度 $\boldsymbol{v}_q(t+\Delta t)$ 的计算需要前一时刻的速度 $\boldsymbol{v}_q(t)$ 和前一时刻的作用力 $\boldsymbol{F}_q(t)$。最后，对 $\boldsymbol{s}_q(t+\Delta t)$ 的计算需要 $\boldsymbol{s}_q(t)$ 和 $\boldsymbol{v}_q(t)$。各个任务之间的通信如图 6-3 所示。这个图清晰地指出，绝大多数任务之间的通信都发生在与单个粒子有关的任务中，所以如果我们将对 $\boldsymbol{s}_q(t)$、$\boldsymbol{v}_q(t)$ 和 $\boldsymbol{F}_q(t)$ 的计算凝聚在一个任务中，任务间的通信会大大简化（见图 6-4）。现在任务与粒子相对应，我们在图 6-4 中对任务间的通信数据进行了标记。例如，在时间 $t$ 时，由粒子 $q$ 指向粒子 $r$ 的箭头用粒子 $q$ 的位置 $\boldsymbol{s}_q$ 标记。

对于简化算法，粒子内部的通信相同。也就是说，为了计算 $\boldsymbol{s}_q(t+1)$，需要 $\boldsymbol{s}_q(t)$ 和 $\boldsymbol{v}_q(t)$；为了计算 $\boldsymbol{v}_q(t+\Delta t)$，需要 $\boldsymbol{v}_q(t)$ 和 $\boldsymbol{F}_q(t)$。因此，我们有必要将与单个粒子相关的计算聚集在一个复合任务中。

在简化算法中，我们利用了作用力的性质 $\boldsymbol{f}_{rq} = -\boldsymbol{f}_{qr}$。所以如果 $q$ 小于 $r$，那么从任务 $r$ 到任务 $q$ 的通信与在基本算法中的通信相同——为了计算 $\boldsymbol{F}_q(t)$，任务 $q$ 需要从任务 $r$ 获得 $\boldsymbol{s}_r(t)$。然而，从任务 $q$ 到任务 $r$ 的通信不再是 $\boldsymbol{s}_q(t)$，而是粒子 $r$ 作用在粒子 $q$ 上的作用力，即 $\boldsymbol{f}_{qr}$。见图 6-5。

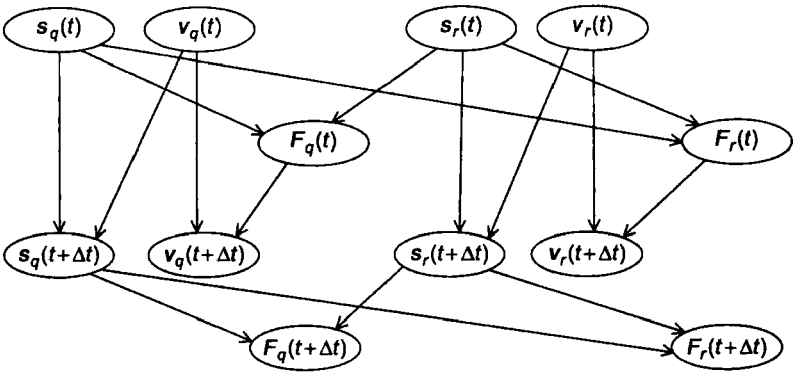



图 6-3　基本 $n$ 体算法中任务之间的通信

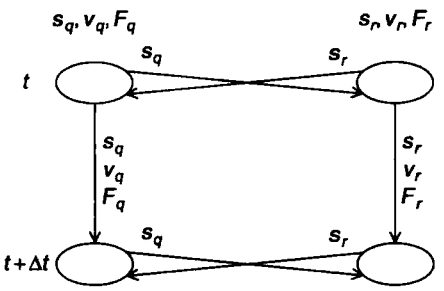



图 6-4　基本 $n$ 体算法中复合任务之间的通信

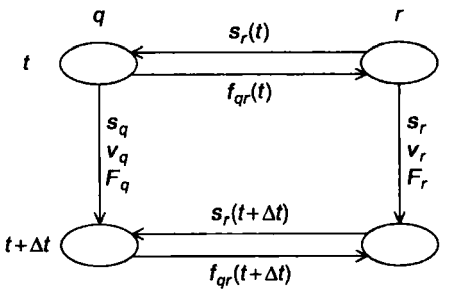



图 6-5　简化 n 体算法中复合任务之间的通信

Foster 方法的最后一步是映射。如果有 $n$ 个粒子和时间步长 $T$，那么不论在基本算法中还是在简化算法中都会有 $nT$ 个任务。在天文学中的 $n$ 体问题往往涉及成千上万个粒子，所以 $n$ 很有可能比可用的核的个数大几个数量级，同时 $T$ 也可能远远大于可用的核的个数。所以，原则上当将任务映射到核时，采取二维的方式。然而，如果考虑欧拉方法的实质，我们将发现将不同时刻与单个粒子相关的任务分配到不同核的方式不是高效的。在估算 $s_q(t+\Delta t)$ 和 $v_q(t+\Delta t)$ 时，欧拉方法必须要先知道 $s_q(t)$、$v_q(t)$ 和 $a_q(t)$。因此，如果将粒子 $q$ 在时间 $t$ 时的任务分配给核 $c_0$，而将粒子 $q$ 在时间 $t+\Delta t$ 时的任务分配给核 $c_1 \neq c_0$，那么必须从 $c_0$ 向 $c_1$ 传递 $s_q(t)$、$v_q(t)$ 和 $F_q(t)$。当然，如果将粒子 $q$ 在时间 $t$ 和时间 $t+\Delta t$ 时的任务映射到同一个核，那么上述的通信就是不必要的了，所以一旦将粒子 $q$ 在前一个时间的任务映射到核 $c_0$，我们最好将粒子 $q$ 下一个时刻的任务也映射到该核上，因为我们不会同时执行粒子 $q$ 在两个不同时刻的任务。因此，将任务映射到核的过程实际上是将粒子分配给核的过程。

乍一看起来，按照每个核大约 $n$/thread_count 个粒子的方式，将粒子分配给核就可以使各个核的负载平衡。这个想法对于基本算法确实成立，因为在基本算法中计算每个核的位置、速度和作用力的工作都是相同的。然而，在简化算法中计算作用力时，计算较小编号的迭代所需要的计算量远远大于较大编号的迭代所需要的计算量。为了说明这一点，我们再来看看简化算法中计算粒子 $q$ 的总作用力的伪代码：

```
for each particle k > q {
   x_diff = pos[q][X] - pos[k][X];
   y_diff = pos[q][Y] - pos[k][Y];
   dist = sqrt(x_diff*x_diff + y_diff*y_diff);
   dist_cubed = dist*dist*dist;
   force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
   force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;
```

```
      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
```

例如，当 $q=0$ 时，我们通过 **for** each particle k > q 做 $n-1$ 次迭代；而当 $q=n-1$ 时，我们将不做任何迭代。因此，对于简化算法，我们相信对粒子进行循环划分比块划分能更均衡地分配计算任务。

然而，在共享内存环境中，循环地将粒子划分给各个核肯定比块划分导致更多的缓存缺失；而在分布式内存环境下，循环法导致的通信数据负载可能比块划分法导致的通信数据负载大（见习题 6.8 和习题 6.9）。

因此，对于将与单个粒子有关的计算组合在一个复合任务的模拟计算方法，我们有如下结论：

（1）$n$ 体问题的基本算法采用块划分法会有更好的性能。

（2）对于简化算法，循环划分可以使得在计算作用力时的负载更为均衡。然而采用该划分方法带来的提升需要与共享内存环境下缓存性能的降低、分布式内存环境下额外的通信负载相权衡。

为了确定将任务映射到核上的最佳方法，我们需要做一些实验。

### 6.1.4 关于 I/O

你可能已经注意到，尽管 I/O 问题在所有的串行算法中都处于重要地位，但关于并行化 $n$ 体问题算法的讨论并没有涉及 I/O 问题。我们在之前的一些章节中多次讨论了 I/O 问题。不同并行系统的 I/O 能力差别很大，那些比较常用的基本 I/O 机制很难获取较高的性能。基本 I/O 机制是为单线程或者单进程程序设计的，当多个线程或者多个进程试图访问 I/O 缓冲区时，系统不会调度它们的访问。例如，如果多个线程试图同时执行：

```
280   printf("Hello from thread %d of %d\n", my_rank, thread_count);
```

那么输出的顺序是不可预测的。更严重的是，同一个线程的输出不是出现在同一行上，而是被来自其他线程的输出分割为多个片段。

因此，正如我们之前所提到的那样，除非是调试程序的输出，否则我们一般认为只有一个线程/进程完成所有的 I/O 操作，并且当计算程序的运行时间时，我们使用这个选项打印最终的时间。此外，我们不会将这个输出操作计算到运行时间中。

当然，即使忽略 I/O 操作的开销，我们也不能忽略它的存在。我们将在讨论并行程序实现的细节时简单讨论它的实现。

### 6.1.5 用 OpenMP 并行化基本算法

我们应该怎么使用 OpenMP 将 $n$ 体问题基本算法中的任务/粒子映射到核上呢？让我们先研究一下串行程序的伪代码：

```
for each timestep {
  if (timestep output) Print positions and velocities of particles;
  for each particle q
     Compute total force on q;
  for each particle q
     Compute position and velocity of q;
}
```

代码中的两个内层循环都是按照粒子进行迭代的。所以理论上，并行化这两个内层 **for** 循环会将任务/粒子映射到核上，我们可能尝试像下面这样修改代码：

```
   for each timestep {
      if (timestep output) Print positions and velocities of
          particles;
#     pragma omp parallel for
      for each particle q
         Compute total force on q;
#     pragma omp parallel for
      for each particle q
         Compute position and velocity of q;
   }
```

事实是，这段代码会进行很多次线程派生和合并操作，这个情况我们不会喜欢。但是在处理这个问题前，让我们先来查看循环本身：我们需要知道是否存在由循环依赖引起的竞争状态。

在基本算法中，第一个循环的形式如下：

```
#  pragma omp parallel for                                          281
   for each particle q {
      forces[q][X] = forces[q][Y] = 0;
      for each particle k != q {
         x_diff = pos[q][X] - pos[k][X];
         y_diff = pos[q][Y] - pos[k][Y];

      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

因为 **for** each particle q 循环的迭代在线程之间，所以对于任意的粒子 $q$，只有一个线程可以访问 forces[q]。不同的线程都可以访问 pos 数组和 mass 数组的相同元素。然而，在循环中这些数组是只读的。其余变量用于内部循环的迭代的临时存储，并且它们可以是私有的。因此，并行化基本算法的第一个循环不会带来任何竞争状态。

第二个循环的形式如下：

```
#  pragma omp parallel for
   for each particle q {
      pos[q][X] += delta_t*vel[q][X];
      pos[q][Y] += delta_t*vel[q][Y];
      vel[q][X] += delta_t/masses[q]*forces[q][X];
      vel[q][Y] += delta_t/masses[q]*forces[q][Y];
   }
```

此时，对于任意的粒子 $q$ 只有一个线程访问 pos [q]、vel [q]、masses [q] 和 forces [q]，标量只能读，所以并行化这个循环也不会带来任何竞争状态。

让我们回到重复派生和合并线程的问题。在伪代码中，有

```
   for each timestep {
      if (timestep output) Print positions and velocities of
          particles;
#     pragma omp parallel for
      for each particle q
         Compute total force on q;
#     pragma omp parallel for
      for each particle q
         Compute position and velocity of q;
   }
```

我们在并行奇偶排序算法中也遇到了类似的问题（见 5.6.2 节）。在那个例子中，我们在最外层的循环前使用了 parallel 指令，而对内层循环使用了 OpenMP 的 **for** 指令。那么，相同的策略在这里会不会起作用呢？也就是说，我们可不可以像下面这样修改代码呢？

282
```
#  pragma omp parallel
   for each timestep {
      if (timestep output) Print positions and velocities of
          particles;
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

这样的修改会在两个 **for** each particle 循环上产生我们想要的效果：同一组中的线程会被这两个内层循环和外层循环的迭代都使用到。然而，我们肯定会在输出语句上遇到问题：按照现在的形式，每个线程都将打印位置和速度，而我们只想要一个线程做 I/O 操作。OpenMP 为这种情况提供了 single 指令：有一组线程执行某个代码块，但是这个代码块中的某一部分只能被这些线程其中的一个执行。加入 single 指令后，我们得到如下的伪代码：

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) {
#        pragma omp single
         Print positions and velocities of particles;
      }
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

修改后还有一些问题需要解决，其中最重要的是当从一条语句转向另一条语句时，可能带来竞争状态。例如，假定线程 0 在线程 1 之前完成了第一个 **for** each particle 循环，接着它在第二个 **for** each particle 循环中更新了分配给它的粒子的位置和速度。显然，这导致线程 1 在第一个 **for** each particle 循环中使用了更新过的位置。然而，需要注意到：在每个用 **for** 指令并行化的结构化块的末尾都有一条隐含的路障，所以如果线程 0 在线程 1 前完成了第一个内部循环，那么它将被阻塞，直到线程 1（和其他所有的线程）完成了第一个内部循环，并且在其他所有的线程执行完第一个内部循环前，它不会开始执行第二个内部循环。这个机制也使得没有哪个线程执行得太快，以至于在其他线程执行完第二个内层循环前就输出了位置和速度。

在 single 指令后也有隐含的路障，尽管在这个程序中路障是不必要的。因为输出语句不会更新任何内存地址，所以其他线程可以在输出完成前继续执行下一个迭代。此外，第一个 **for** 内层循环只是更新了 forces 数组，所以它不会导致执行输出语句的线程打印不正确的值。因为第
283 一个内层循环末尾的路障，所以没有一个线程可以在输出结束之前就开始执行第二个内层循环更新位置和速度。因此我们可以用 nowait 子句修改 single 指令。如果 OpenMP 指令的实现支持这个子句，那么它仅仅会解除 single 指令默认的路障。这个子句也适用于 **for**、parallel **for** 和 parallel 指令。需要注意的是，此时使用 nowait 子句不会在性能上带来太大的性能提升，因为这两个 **for** each particle 循环有默认的路障，而默认的路障不会允许某个线程在其他线程前执行太多的语句。

最后，为了确保程序迭代采取块划分，需要在每个 **for** 指令之前加上一条调度子句：

```
#     pragma omp for schedule(static, n/thread_count)
```

### 6.1.6　用 OpenMP 并行化简化算法

简化算法增加了一个额外的内部循环，将 forces 数组中的元素置为 0。如果对简化算法采取相同的并行化方法，应当用 **for** 指令并行化这个循环。如果这样做，会发生什么呢？也就是说，如果通过下面的伪代码来并行化简化算法会产生什么样的结果？

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) {
#        pragma omp single
         Print positions and velocities of particles;
      }
#     pragma omp for
      for each particle q
         forces[q] = 0.0;
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

因为在迭代之间没有循环依赖，所以对 forces 数组初始化的并行化没有什么问题。而简化算法对位置和速度的更新与基本算法相同，所以如果对 forces 的计算是正确的，那么整个算法应当是正确的。

上面的并行化会怎样影响计算 forces 循环的正确性呢？在简化算法中，循环有如下的形式：

```
#  pragma omp for /* Can be faster than memset */                    284
   for each particle q {
      force_qk[X] = force_qk[Y] = 0;
      for each particle k > q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
      force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
}
```

和以前一样，我们感兴趣的变量是 pos、masses 和 forces，由于其他变量只会在单个迭代中访问，因此是私有变量。此外，pos 和 masses 数组中的元素是只读的，不能更新。因此，我们只需要研究 forces 数组的元素。与基本算法不同，在简化算法中，一个线程可能更新 forces 数组中不属于分配给该线程粒子的元素。例如，假设有两个线程和四个粒子，并且对粒子采用了块划分，那么粒子 3 所受的总作用力是：

$$\boldsymbol{F}_3 = -\boldsymbol{f}_{03} - \boldsymbol{f}_{13} - \boldsymbol{f}_{23}$$

此外，线程 0 要计算 $\boldsymbol{f}_{03}$ 和 $\boldsymbol{f}_{13}$，而线程 1 要计算 $\boldsymbol{f}_{23}$。因此，对 forces[3]的更新肯定会产生竞争状态。一般而言，对 forces 数组元素的更新会导致竞争状态。

这个问题的一个比较直观的解决方案是使用 critical 指令限制对 forces 数组元素的访问。至少有好几种方式可以达到该目的，而最简单的方法是将 critical 指令放在对 forces 数

组更新之前：

```
#  pragma omp critical
   {
      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
```

然而，这种方式会使得对 forces 数组元素的访问被串行化了。一次只能更新 forces 数组中的一个元素，而对临界区的争用很有可能会导致程序的性能急剧地下降。见习题 6.3。

285 另外一种可行的方法是针对每一个粒子建立一个临界区。然而，如我们所看到的，OpenMP 还不能很好地支持临界区数目的变化，所以需要给每个粒子加锁，加锁后的代码如下：

```
omp_set_lock(&locks[q]);
forces[q][X] += force_qk[X];
forces[q][Y] += force_qk[Y];
omp_unset_lock(&locks[q]);

omp_set_lock(&locks[k]);
forces[k][X] -= force_qk[X];
forces[k][Y] -= force_qk[Y];
omp_unset_lock(&locks[k]);
```

这段代码假定主线程已经创建了一个共享锁数组，其中每个粒子一个锁，而当更新 forces 数组元素时，我们首先要获得对应粒子的锁。尽管这个方法已经比单个临界区的性能提升了很多，但是还是比不上串行程序的性能。见习题 6.4。

另外一种可行的解决方案是将对作用力的计算分成两个阶段。在第一个阶段中，每个线程所进行的计算与错误的并行程序中的计算相同；而不同是，现在计算的结果都存储在它自己的作用力数组中。然后，在第二个阶段中，与粒子 $q$ 相对应的线程将加上被其他线程计算出的该粒子所受作用力的部分。在上面的例子中，线程 0 要计算 $-f_{03}$ $-f_{13}$，而线程 1 要计算 $-f_{23}$。在每个线程都计算出分配给它的作用力部分时，对应于粒子 3 的线程 1 通过将这两个值相加得到粒子 3 所受的总作用力。

让我们看看更大一些的例子。假定有三个线程和六个粒子。如果我们对粒子采取块划分，那么第一个阶段的计算如表 6-1 所示。从表中最后三列可以看出每个线程在计算总作用力中的作用。在第二个阶段的计算中，表中第一列的线程将对它们所分配到的行求和，以得到相应粒子所受的作用力之和。

需要注意的是，对粒子块划分时，没有进行什么特别的修改。表 6-2 展示了在对粒子采用循环划分时的计算过程。将表 6-2 和表 6-1 相比，我们发现循环划分在负载平衡方面做得更好。

**表 6-1　块划分的简化算法在第一个阶段的计算**

| 线程 | 粒子 | 线程 | | |
|---|---|---|---|---|
| | | 0 | 1 | 2 |
| 0 | 0 | $f_{01}+f_{02}+f_{03}+f_{04}+f_{05}$ | 0 | 0 |
| | 1 | $-f_{01}+f_{12}+f_{13}+f_{14}+f_{15}$ | 0 | 0 |
| 1 | 2 | $-f_{02}+f_{12}$ | $f_{23}+f_{24}+f_{25}$ | 0 |
| | 3 | $-f_{03}-f_{13}$ | $-f_{23}+f_{34}+f_{35}$ | 0 |
| 2 | 4 | $-f_{04}-f_{14}$ | $-f_{24}-f_{34}$ | $f_{45}$ |
| | 5 | $-f_{05}-f_{15}$ | $-f_{25}-f_{35}$ | $-f_{45}$ |

表 6-2　循环划分的简化算法在第一个阶段的计算

286

| 线程 | 粒子 | 线程 | | |
|---|---|---|---|---|
| | | 0 | 1 | 2 |
| 0 | 0 | $f_{01}+f_{02}+f_{03}+f_{04}+f_{05}$ | 0 | 0 |
| 1 | 1 | $-f_{01}$ | $f_{12}+f_{13}+f_{14}+f_{15}$ | 0 |
| 2 | 2 | $-f_{02}$ | $-f_{12}$ | $f_{23}+f_{24}+f_{25}$ |
| 0 | 3 | $-f_{03}+f_{34}+f_{35}$ | $-f_{13}$ | $-f_{23}$ |
| 1 | 4 | $-f_{04}-f_{34}$ | $-f_{14}+f_{45}$ | $-f_{24}$ |
| 2 | 5 | $-f_{05}-f_{35}$ | $-f_{15}-f_{45}$ | $-f_{25}$ |

修改后的算法在第一个阶段与之前的算法相同，只是每个线程将计算出的作用力加到它自己的子数组 loc_forces 中：

```
#  pragma omp for
   for each particle q {
      force_qk[X] = force_qk[Y] = 0;
      for each particle k > q {
         x_diff = pos[q][X] - pos[k][X];
         y_diff = pos[q][Y] - pos[k][Y];
         dist = sqrt(x_diff*x_diff + y_diff*y_diff);

      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
      force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

      loc_forces[my_rank][q][X] += force_qk[X];
      loc_forces[my_rank][q][Y] += force_qk[Y];
      loc_forces[my_rank][k][X] -= force_qk[X];
      loc_forces[my_rank][k][Y] -= force_qk[Y];
   }
}
```

在第二个阶段中，每个线程加上其他线程计算出的作用力，得出分配给它的粒子所受的总作用力：

```
#  pragma omp for
   for (q = 0; q < n; q++) {
      forces[q][X] = forces[q][Y] = 0;
      for (thread = 0; thread < thread_count; thread++) {
         forces[q][X] += loc_forces[thread][q][X];
         forces[q][Y] += loc_forces[thread][q][Y];
      }
   }
```

在继续其他工作之前，我们必须确定我们没有不小心地引入了新的竞争状态。在第一个阶段中，287
因为每一个线程都是写自己的子数组，所以对 loc_forces 的更新不会引入竞争状态。此外，在第二个阶段中，只有粒子 $q$ 的所有者线程 $q$ 才会写 forces[q]，所以在第二个阶段中也不会引入竞争状态。最后，由于每个并行化的 **for** 循环的都有隐含的路障，因此我们不用担心有某个线程执行得太快，以至于它会用到还没有被正确初始化的变量；或者某个线程运行得太慢，以至于它会用到已经被其他线程修改过的变量。

### 6.1.7　评估 OpenMP 程序

在比较基本算法和简化算法之前，我们需要确定如何调度并行化的 **for** 循环。对于基本算法，我们看到任何将迭代均匀地划分给线程的调度都能够取得较好的计算负载均衡（一般假定只

有一个线程/核)。我们也注意到，对迭代进行块划分将产生比循环划分更少的缓存缺失。因此，我们认为块调度是基本算法最好的策略。

在简化算法中，随着迭代的增加，第一阶段对作用力的计算逐渐减少，因此循环划分能更好地将计算均等地分配给每个线程。在剩下的并行 **for** 循环中（对 `loc_forces` 数组的初始化、作用力的第二阶段计算、位置和速度的更新）每次迭代所要做的工作量大致相同。因此，断章取义地看，这些循环采用块划分时性能会更好。然而，对一个循环的调度会影响另外一个循环的性能（见习题 6.10），所以如果对一个循环采用了循环调度，而对其他循环采取块调度可能会降低程序的性能。

表 6-3 显示了在这些策略下，没有进行 I/O 操作的 $n$ 体程序在我们系统上的性能。在测试中，有 400 个粒子和 1000 个时间点。“Default Sched” 列给出了当所有的内部循环都采用默认块调度时，OpenMP 简化算法的运行时间。“Forces Cyclic” 列给出了第一阶段作用力计算采用循环调度，而其他内部循环采用默认调度时的程序运行时间。“All Cyclic” 列给出了所有的内部循环都采用循环调度时，程序的运行时间。串行算法的运行时间与单线程运行算法的时间差小于 1%，所以没有在表 6-3 中列出。

**表 6-3 用 OpenMP 并行化 $n$ 体问题算法的运行时间**

| 线程数 | 基本算法 | 简单算法 | 简单算法 | 简单算法 |
|---|---|---|---|---|
| | | Default Sched | Forces Cyclic | All Cyclic |
| 1 | 7.71 | 3.90 | 3.90 | 3.90 |
| 2 | 3.87 | 2.94 | 1.98 | 2.01 |
| 4 | 1.95 | 1.73 | 1.01 | 1.08 |
| 8 | 0.99 | 0.95 | 0.54 | 0.61 |

288 注意，使用多个线程运行简化算法时，默认调度比循环调度的运行时间要长 50% ~75%。在这种情况下，显然使用循环调度比使用默认调度更好，负载平衡所带来的性能提升可以较好地弥补缓存缺失带来的开销。

对于由两个线程运行的简化算法，只是将作用力计算的第一个循环采用循环法的调度策略与所有的循环都采用循环法的调度策略相比较，性能相差不大。然而，当增加线程数目后，对所有循环都采用循环调度策略的性能开始下降。在这个例子中，当使用更多的线程运行程序时，由负载不均衡引起的开销要小于伪共享引起的开销。

最后，基本算法比在作用力计算时采用循环调度的简化算法需要多花费两倍的运行时间，所以当有较大的内存空间可用时，简化算法显然要更好一些。然而，为了存储总作用力，简化算法额外需要 `thread_count` 倍的内存空间，所以当粒子的数目相当大时，使用简化算法变得不可行。

### 6.1.8 用 Pthreads 并行化算法

用 Pthreads 并行化 $n$ 体问题的两个算法与用 OpenMP 并行化这两个算法十分相似，不同之处仅在于实现的细节上，所以我们将指出 Pthreads 与 OpenMP 在实现上主要的不同，而不是重复我们之前的讨论，同时我们也会涉及两者之间一些重要的相似之处。

- 在 Pthreads 中，缺省情况下局部变量是私有的，而所有的共享变量都是全局的。
- Pthreads 中的主要数据结构与 OpenMP 中的数据结构相同：向量由二维 `double` 型数组表示，而粒子的质量、位置和速度都存储在结构体中。粒子所受的作用力存储在向量数

组中。

- 启动 Pthreads 基本上与启动 OpenMP 相同：主线程获取命令行参数，分配和初始化主要的数据结构。
- Pthreads 实现与 OpenMP 实现的不同主要体现在对内部循环并行化的细节上。因为 Pthreads 中没有类似于 `parallel for` 的指令，所以必须显式地决定哪个循环变量对应于线程的计算。为了方便，编写函数 `Loop_schedule` 来决定：
  - 循环变量的初始值
  - 循环变量的最终值
  - 循环变量的增量

  289

  该函数的输入是：
  - 调用该函数的线程编号
  - 线程的数目
  - 总的迭代数
  - 指明采用块调度还是循环调度的参数
- Pthreads 与 OpenMP 实现的另外一个不同之处在于它们的路障。OpenMP 的 `parallel for` 指令有隐含的路障，而这是十分重要的。例如，我们不想某个线程在所有线程完成对作用力的计算之前就开始更新位置，因为它可能会使用旧的作用力数据，而其他线程则会使用旧的位置信息。如果在 Pthreads 实现中，简单地按照线程划分循环的迭代，那么在内部 `for` 循环的末尾不会有路障，由此会产生竞争状态。因此，必须在内部循环可能产生竞争状态的地方显式地使用路障。Pthreads 标准中包含了路障，但是有些系统并没有实现它，所以我们定义了一个函数，通过 Pthreads 条件变量实现路障。细节请见 4.8.3 节。

### 6.1.9　用 MPI 并行化基本算法

将与单个粒子相关的计算视为一个复合任务，使得用 MPI 并行化基本算法十分直观。任务之间的通信仅发生在计算作用力时，每个任务/粒子需要获取其他粒子的位置和质量。`MPI_Allgather` 函数就是专门为这种情况设计的，该指令为每个线程从其他线程收集同样的信息。已经发现：采用块划分时程序的性能最好，所以采用块划分将粒子映射到进程上。

在共享内存的实现中，我们将与单个粒子相关的信息（质量、位置、速度）放在一个结构体中。然而，如果在 MPI 的实现中使用这个数据结构，就需要在调用 `MPI_Allgather` 时使用派生数据类型，而使用派生数据类型的通信往往比使用基本 MPI 数据类型的通信慢。因此，有必要使用单独的数组分别存储质量、位置和速度，并且还需要将所有粒子的位置存储在一个单独的数组中。如果每一个进程有足够的内存，那么这些变量都可以存储在不同的数组中。实际上，如果内存足够，每个进程可以存储整个质量数组，因为质量不会改变，而且它们的值只会在初始化的时候传递。

另一方面，如果内存空间有限，那么有些 MPI 集合通信可以使用“in-place”选项。在我们的例子中，假设数组 `pos` 是可以存储 $n$ 个粒子的位置，`vect_mpi_t` 是一个存储两个连续 `double` 型的 MPI 数据类型，$n$ 可以被 `comm_sz` 整除，且 `loc_n = n/comm_sz`。那么，将局部的位置信息存储在独立的数组 `loc_pos` 中，可以调用 `MPI_Allgather` 从进程中收集所有的位置信息：

```
MPI_Allgather(loc_pos, loc_n, vect_mpi_t,
      pos, loc_n, vect_mpi_t, comm);
```

如果不能为 `loc_pos` 提供额外的存储空间，则可以让进程 $q$ 将它的本地位置存储在 `pos` 中的第 $q$ 个位置。也就是说，每个进程的本地位置应该存储在每个进程的 `pos` 数组的相应块中。

290

```
Process 0:  pos[0], pos[1], . . . . , pos[loc_n-1]
Process 1:  pos[loc_n], pos[loc_n+1], . . . . , pos[loc_n + loc_n-1]
   . . .
Process q:  pos[q*loc_n], pos[q*loc_n+1], . . . . , pos[q*loc_n +
    loc_n-1]
   . . .
```

通过这种方式初始化每个进程的 pos 数组，就可以按照如下方式调用 MPI_Allgather：

```
MPI_Allgather(MPI_IN_PLACE, loc_n, vect_mpi_t,
  pos, loc_n, vect_mpi_t, comm);
```

在这个调用中，忽略第一个 loc_n 和 vect_mpi_t 参数，然而保留它们可以增加程序的可读性。

在给出的程序中，对数据结构做出以下设定：

- 每个进程存储整个包含所有粒子质量的全局数组。
- 每个进程仅使用一个 $n$ 个元素的数组来存储位置信息。
- 每个进程通过指针 loc_pos 指向 pos 的起始地址。因此，对于进程 0，local_pos = pos；对于进程 1，local_pos = pos + loc_n，以此类推。

根据这些设定，按照程序 6-2 所示的伪代码实现基本算法。进程 0 读取并广播命令行参数，同时它读取输入并打印结果。在第一行，进程 0 需要传递输入，因此 Get input data 可以按如下方法实现：

```
if (my_rank == 0) {
   for each particle
      Read masses[particle], pos[particle], vel[particle];
}
MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
MPI_Bcast(pos, n, vect_mpi_t, 0, comm);
MPI_Scatter(vel, loc_n, vect_mpi_t, loc_vel, loc_n, vect_mpi_t, 0,
    comm);
```

所以，进程 0 将读取所有的初始条件，并存入 $n$ 元数组中。因为在每个进程中，要存储所有粒子的质量，所以 masses 将被广播。pos 也将被广播，因为每个进程需要在主 **for** 循环进行第一个阶段的作用力计算，需要全局的位置数组。然而，速度仅在本地使用，用来更新位置和速度，所以只是把 vel 散布到各个进程中。

291

**程序 6-2　$n$ 体问题基本算法的 MPI 实现伪代码**

```
 1  Get input data;
 2  for each timestep {
 3     if (timestep output)
 4        Print positions and velocities of particles;
 5     for each local particle loc_q
 6        Compute total force on loc_q;
 7     for each local particle loc_q
 8        Compute position and velocity of loc_q;
 9     Allgather local positions into global pos array;
10  }
11  Print positions and velocities of particles;
```

注意，在程序 6-2 的第 9 行外部 **for** 循环的末尾，收集更新过的位置信息，这确保了位置信息在第 4 行和第 11 行可以被访问。如果要打印每一个时间点的结果，计算作用力之前需要先调用 MPI_Allgather，并让进程 0 收集所有粒子的位置信息，那么可以消除一个代价较高的集合通信调用。当外层 **for** 循环按照如上所述组织时，可以按照下面的伪代码实现输出：

```
Gather velocities onto process 0;
if (my_rank == 0) {
   Print timestep;
   for each particle
      Print pos[particle] and vel[particle]
}
```

### 6.1.10　用 MPI 并行化简化算法

直接实现简化算法会十分复杂。在计算作用力前，每个进程需要收集位置的子集，而在作用力计算后，每个进程将传递它计算出的作用力，并与它接收到的作用力求和。图 6-6 显示了在三个进程、六个粒子，并使用块划分将粒子分配给进程的条件下，进程之间的通信情况。可以预见，当采用循环调度时，通信会变得更加复杂（见习题 6.13）。当然这些通信是可以实现的。然而，除非在实现时相当小心，否则程序会变得很慢。 292

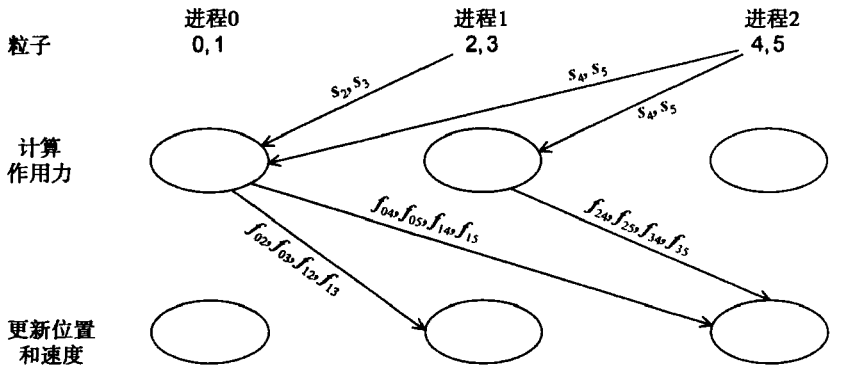



图 6-6　$n$ 体问题简化算法可能的 MPI 实现中的通信

幸运的是，可以采用另外一种更加简单的方式——称为**环形传递**（ring pass）的通信结构。在环形传递中，进程可以看做是用一个环相互连接（见图 6-7）。进程 0 直接与进程 1 和进程 comm_sz - 1 通信，而进程 1 与进程 0 和进程 2 通信，以此类推。在环形传递中的通信分阶段进行，在每一个阶段中，每个进程向标号较低的相邻进程传送数据，从标号较高的进程接收数据。因此，进程 0 可以向进程 comm_sz - 1 传递数据，从进程 1 接收数据；进程 1 从进程 2 接收数据，而向进程 0 传递数据，以此类推。总的来说，进程 $q$ 向进程 $(q-1+\text{comm\_sz})\%\ \text{comm\_sz}$ 传递数据，从进程 $(q+1)\%\ \text{comm\_sz}$ 接收数据。

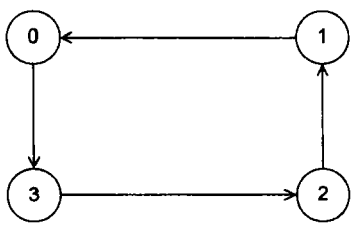



图 6-7　进程组成的环

通过在环结构上重复地接收和传递数据，可以使得每个进程都能够获取所有粒子的位置信息。在第一个阶段，每个进程都向标号较小的相邻进程传递分配给它的粒子的位置信息，从标号较高的相邻进程接收分配给该进程的粒子的位置信息。在下一个阶段中，每个进程都将传递在第一个阶段接收到的位置信息。这个过程会重复 comm_sz - 1 次，直到所有的进程都获知所有粒子的位置信息。图 6-8 显示了在有 4 个进程、8 个粒子，采取循环划分的情况下，三个阶段的情况。

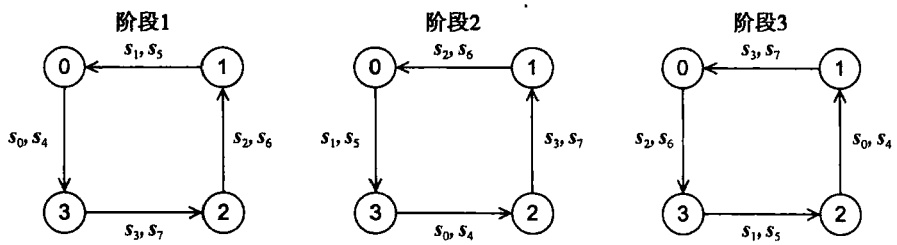



图 6-8　位置信息在环上的传递

293 当然，简化算法的好处是，由于 $f_{kq} = -f_{qk}$，所以我们不需要计算所有粒子对 $q$ 和 $k$ 之间的作用力。为了利用这一点，在早先使用的简化算法中，粒子间的作用力可以分为加到粒子总作用力中去的和从粒子总作用力中减去的两种。例如，假设有 6 个粒子，那么简化算法将按如下方法计算粒子 3 所受的总作用力：

$$F_3 = -f_{03} - f_{13} - f_{23} + f_{34} + f_{35}$$

环形传递方式计算作用力的关键是：从总作用力中减去的作用力是由其他任务/粒子计算的，而加到总作用力中的作用力由该任务/粒子本身计算。因此，粒子 3 所受到的粒子间的作用力的计算任务分配如下：

| 作用力 | $f_{03}$ | $f_{13}$ | $f_{23}$ | $f_{34}$ | $f_{35}$ |
|---|---|---|---|---|---|
| 任务/粒子 | 0 | 1 | 2 | 3 | 3 |

所以，假设在环形传递中，不仅传递 loc_n = $n$ /comm_sz 个粒子的位置信息，同时也传递 loc_n 个粒子的作用力，那么在每一个阶段，一个进程可以：

（1）计算分配给它的粒子与它接收到位置信息的粒子间的作用力。

（2）一旦粒子间的作用力被计算出来了，进程就可以将这些作用力加到相应粒子的局部作用力数组中，并且减去接收到的粒子间的作用力。

更多细节和可行方案请见 [15, 34]。

现在研究当有 4 个粒子、2 个进程，并且采用循环法分配粒子时计算是如何进行的（见表 6-4）。loc_pos 数组和 loc_forces 数组分别存储的是局部的位置和作用力信息，这些信息不会在进
294 程之间传递。需要在进程之间传递的数组是 tmp_pos 和 tmp_forces。

**表 6-4　在环形传递中的作用力计算**

| 时间 | 变量 | 进程 0 | 进程 1 |
|---|---|---|---|
| 开始 | loc_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | loc_forces | 0, 0 | 0, 0 |
| | tmp_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | tmp_forces | 0, 0 | 0, 0 |
| 作用力计算后 | loc_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | loc_forces | $f_{02}$, 0 | $f_{13}$, 0 |
| | tmp_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | tmp_forces | 0, $-f_{02}$ | 0, $-f_{13}$ |
| 第一次通信后 | loc_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | loc_forces | $f_{02}$, 0 | $f_{13}$, 0 |
| | tmp_pos | $s_1$, $s_3$ | $s_0$, $s_2$ |
| | tmp_forces | 0, $-f_{13}$ | 0, $-f_{02}$ |
| 作用力计算后 | loc_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | loc_forces | $f_{01}+f_{02}+f_{03}$, $f_{23}$ | $f_{12}+f_{13}$, 0 |
| | tmp_pos | $s_1$, $s_3$ | $s_0$, $s_2$ |
| | tmp_forces | $-f_{01}$, $-f_{03}-f_{13}-f_{23}$ | 0, $-f_{02}-f_{12}$ |
| 第二次通信后 | loc_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | loc_forces | $f_{01}+f_{02}+f_{03}$, $f_{23}$ | $f_{12}+f_{13}$, 0 |
| | tmp_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | tmp_forces | 0, $-f_{02}-f_{12}$ | $-f_{01}$, $-f_{03}-f_{13}-f_{23}$ |
| 作用力计算后 | loc_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | loc_forces | $f_{01}+f_{02}+f_{03}$, $-f_{02}-f_{12}+f_{23}$ | $-f_{01}+f_{03}-f_{13}-f_{23}$ |
| | tmp_pos | $s_0$, $s_2$ | $s_1$, $s_3$ |
| | tmp_forces | 0, $-f_{02}-f_{12}$ | $-f_{01}$, $-f_{03}-f_{13}-f_{23}$ |

在环形传递开始前，存储位置的数组都用本地粒子的位置进行初始化，且存储作用力的数组中的元素都置为0。在环形传递开始前，每个进程首先要计算分配给它的粒子间的作用力。进程0计算$f_{02}$，进程1计算$f_{13}$。这些值将加到loc_forces中对应的位置，并从tmp_forces合适的位置中减去这些值。

现在两个进程相互交换tmp_pos和tmp_forces，并计算分配给它们的粒子和接收到的粒子间的作用力。在简化算法中，标号较低的任务/粒子负责计算。进程0计算$f_{01}$、$f_{03}$和$f_{23}$，而进程1计算$f_{12}$。与之前一样，计算出的作用力加到loc_forces合适的地址中，并从tmp_forces合适的地址中减去。

为了完成算法，需要在最后交换tmp数组[⊖]。每个进程收到已更新的tmp_forces后，可以做一个简单的向量求和运算

```
loc_forces += tmp_forces
```

以完成计算。 295

**程序6-3 $n$体问题简化算法的MPI实现伪代码**

```
 1  source = (my_rank + 1) % comm_sz;
 2  dest = (my_rank - 1 + comm_sz) % comm_sz;
 3  Copy loc_pos into tmp_pos;
 4  loc_forces = tmp_forces = 0;
 5
 6  Compute forces due to interactions among local particles;
 7  for (phase = 1; phase < comm_sz; phase++) {
 8     Send current tmp_pos and tmp_forces to dest;
 9     Receive new tmp_pos and tmp_forces from source;
10     /* Owner of the positions and forces we're receiving */
11     owner = (my_rank + phase) % comm_sz;
12     Compute forces due to interactions among my particles
13        and owner's particles;
14  }
15  Send current tmp_pos and tmp_forces to dest;
16  Receive new tmp_pos and tmp_forces from source;
```

因此，我们可以使用环形传递按照程序6-3中的伪代码实现简化算法中对作用力的计算。需要注意的是：按照MPI的说明，在第8行、第9行、第15行、第26行使用MPI_Send和MPI_Recv接收和发送信息是不安全的，因为当系统没有足够的缓存时，它们会被挂起。在这种情况下，MPI提供了MPI_Sendrecv和MPI_Sendrecv_replace。因为使用了相同的内存存储传递的数据和接收的数据，所以可以使用MPI_Sendrecv_replace。

开启一条消息的时间开销是比较大的，所以可以通过使用一个数组同时存储tmp_pos和tmp_forces的方式降低通信代价。例如，可以为数组tmp_data分配可以存储2×loc_n个vec_t类型对象的空间，其中前loc_n个位置存放tmp_pos，而后loc_n个位置存放tmp_forces，然后让指针tmp_pos和tmp_forces分别指向tmp_data[0]和tmp_data[loc_n]。

在实现第12行和第13行对作用力的计算时，主要困难是决定当前进程是否要计算分配给该进程的粒子和接收到位置信息的粒子间的相互作用力。如果研究简化算法（程序6-1），任务/粒子$q$负责计算$f_{qr}$（$q<r$）。然而，loc_pos和tmp_pos（或者同时包含tmp_pos和tmp_forces的数组）使用的是局部下标，而不是全局下标。也就是说，当访问loc_pos中的一个元素时，使用的下标在0～loc_n－1内，而不是在0～$n$－1内，所以我们试图按照如下的伪代码

⊖ 实际上，在最后的通信中我们仅需要交换tmp_forces。

实现对作用力的计算：

296
```
for (loc_part1 = 0; loc_part1 < loc_n-1; loc_part1++)
   for (loc_part2 = loc_part1+1; loc_part2 < loc_n; loc_part2++)
   Compute_force(loc_pos[loc_part1], masses[loc_part1],
         tmp_pos[loc_part2], masses[loc_part2],
         loc_forces[loc_part1], tmp_forces[loc_part2]);
```

我们会碰到几个问题。首先，masses 显然是一个全局数组，而我们用局部下标访问它的元素；其次，local_part1 和 local_part2 的相对大小并不能决定是否应该计算它们之间的作用力，需要用全局下标来决定作用力的计算。例如，有四个粒子，两个进程，前面的代码由进程 0 执行，那么当 local_part1 =0 时，内部循环将从 local_part2 =1 开始而跳过 local_part2 =0；然而，如果采用的是循环调度，local_part1 =0 对应的是全局粒子 0，local_part2 =0 对应的是全局粒子 1，则应该计算这两个粒子间的相互作用力。

显然，问题在于我们使用了局部粒子下标，而不是全局粒子下标。因此，在采用循环调度时，我们可以修改代码，使得循环按照全局粒子下标迭代：

```
for (loc_part1 = 0, glb_part1 = my_rank;
     loc_part1 < loc_n-1;
     loc_part1++, glb_part1 += comm_sz)
  for (glb_part2 = First_index(glb_part1, my_rank, owner, comm_sz),
       loc_part2 = Global_to_local(glb_part2, owner, loc_n);
       loc_part2 < loc_n;
       loc_part2++, glb_part2 += comm_sz)
       Compute_force(loc_pos[loc_part1], masses[glb_part1],
             tmp_pos[loc_part2], masses[glb_part2],
             loc_forces[loc_part1], tmp_forces[loc_part2]);
```

函数 First_index 按照下面的条件确定全局下标 glb_part2：

(1) 粒子 glb_part2 分配给标号为 owner 的进程。

(2) glb_part1 < glb_part2 < glb_part1 + comm_sz。

函数 Global_to_local 将一个全局粒子下标转换为一个局部粒子下标，函数 Compute_force 将计算两个粒子间的相互作用力。我们已经知道如何实现 Compute_force。对这两个函数的实现请见习题 6.15 和习题 6.16。

### 6.1.11 MPI 程序的性能

表 6-5 说明了在 800 个粒子、1000 个时间点的条件下，两个 $n$ 体程序在 Infiniband 连接的集群上的运行时间。所有的计时都是在一个集群节点运行一个进程的情况下得出的。串行程序的运行时间与单进程 MPI 程序的运行时间的差在 1% 以内，所以表 6-5 中没有列出。

297 尽管基本算法达到了更好的效率，但显然简化算法的性能远远好于基本算法。例如，基本算法在 16 个节点上的效率是 0.95，而简化算法在 16 个节点上的性能大约只是 0.70。

表 6-5 $n$ 体问题算法 MPI 实现的性能（时间：秒）

| 进程数 | 基本算法 | 简化算法 |
|---|---|---|
| 1 | 17.30 | 8.68 |
| 2 | 8.65 | 4.45 |
| 4 | 4.35 | 2.30 |
| 8 | 2.20 | 1.26 |
| 16 | 1.13 | 0.78 |

表 6-6　$n$ 体问题算法 OpenMP 和 MPI 实现的性能（时间：秒）

| 进程/线程数 | OpenMP | | MPI | |
|---|---|---|---|---|
| | 基本算法 | 简化算法 | 基本算法 | 简化算法 |
| 1 | 15.13 | 8.77 | 17.30 | 8.68 |
| 2 | 7.62 | 4.42 | 8.65 | 4.45 |
| 4 | 3.85 | 2.26 | 4.35 | 2.30 |

需要强调的一点是，简化 MPI 算法比基本 MPI 算法能更有效地利用内存。基本算法在每个进程中都要为 $n$ 个位置信息提供存储空间，而简化算法只需要为 $n/$comm_sz 个位置和 $n/$comm_sz 个作用力提供额外的存储空间，所以基本算法的每个进程大约要比简化算法的进程多需要 comm/2 倍的存储空间。当 $n$ 和 comm_sz 十分大时，这个因素可以决定是只使用到系统的主存还是不得不使用二级存储来实现模拟计算。

进行性能测试的集群的每个节点有 4 个核，所以可以比较用 OpenMP 实现的性能和用 MPI 实现的性能（见表 6-6）。我们可以看到基本 OpenMP 程序比基本 MPI 程序要快很多，这个结果是显然的，因为 MPI_Allgather 的开销很大。然而，令人惊讶的是，简化的 MPI 程序的性能与 OpenMP 程序的性能相当。

现在分别查看 OpenMP 程序和 MPI 程序运行时所需要的内存。假设有 $n$ 个粒子，$p$ 个进程或线程，每一种解决方案都会为局部位置信息和局部速度信息分配相同的存储空间。MPI 程序为每个进程分配 $n$ 个 **double** 型的存储空间存放质量，$4n/p$ 个 **double** 型的存储空间存放 tmp_pos 和 tmp_forces 数组，所以除了局部的位置信息和速度信息外，MPI 程序为每个进程中还要存储：

298

$$n + 4n/p$$

个 **double** 型变量。OpenMP 程序总共要 $2pn + 2n$ 个 **double** 型变量存储作用力，$n$ 个 **double** 型变量存储质量，所以除了局部的位置和速度信息外，对于每一个线程，OpenMP 程序还需要

$$3n/p + 2n$$

个 **double** 型变量的存储空间。因此 OpenMP 程序比 MPI 程序多

$$n - n/p$$

个 **double** 型变量用于存储局部变量。换句话说，如果 $n$ 很大，OpenMP 程序对局部存储空间的需求远远大于 MPI 程序。所以，对于给定的线程数或进程数，MPI 程序可以比 OpenMP 程序进行更大规模的模拟。当然，出于硬件方面的考虑，可以使用的 MPI 进程数目很有可能比可以使用的 OpenMP 线程数目要大，所以 MPI 程序可以进行的最大规模的模拟比 OpenMP 程序可以进行的最大规模的模拟大得多。MPI 版本的简化算法比其他版本的可扩展性更好，环形传递算法为 $n$ 体问题的设计带来了突破。

## 6.2　树形搜索

许多问题可以归纳为树形搜索问题。举个简单的例子，考虑旅行商（Traveling Salesperson Problem，TSP）问题。在 TSP 问题中，旅行商有一个要访问城市的列表和每两个城市之间旅行的开销，他要访问每个城市仅一次，并且回到出发的城市。现在的问题是如何使这个任务的开销最小。一条以出发城市为起点、访问每个城市仅一次，并且以出发城市为终点的路径叫做*回路*，TSP 问题想要寻找的是代价最小的回路。

不幸的是，旅行商问题已被证明是 **NP 完全**（NP-complete）问题。换句话说，目前还没有哪一个已知的算法可以比穷举法更好地解决该问题。穷举法意味着穷尽每一种可行的方案，然后找

到最优方案。TSP 问题可能的解决方案的数目随着城市数目的增加成指数级增长。例如，在 $n$ 个城市的 TSP 问题中添加一个城市，就会增加 $n-1$ 倍的可行解。因此，尽管对于 4 个城市的 TSP 问题只有 6 个可行解，但 5 个城市的 TSP 问题却有 $4\times6=24$ 个可行解，6 个城市时有 $5\times24=120$ 个可行解，7 个城市有 $6\times120=720$ 个可行解，依此类推。实际上，100 个城市的可行解的数目甚
299 至比宇宙中原子的数目还要多。

如果为 TSP 问题找到一种比穷举法更好的算法，那么对于其他 NP 完全问题也同样可以找到一种更快的方法。不过到目前为止，还没有找到比穷举法更好的 TSP 问题的算法，将来我们也不大可能找到。

所以如何解决 TSP 问题呢？目前有一些比较智能的办法。然而，我们要探讨的是一个十分简单的、类似于树形搜索的算法。基本思想是在搜寻可行解时构造一棵树。树的叶子结点对应于一种回路，树的其他结点对应于部分回路——访问了部分城市，但不是全部城市的路线。

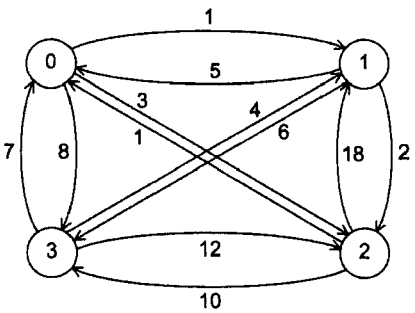


图 6-9　4 个城市的 TSP 问题

树中的每个结点都有代价，也就是部分回路的代价。可以利用这些信息去掉树中的某些结点。因此，需要跟踪目前为止代价最小的部分回路，如果发现一个部分回路不可能导出代价更小的回路时，就不用再扩展该部分回路了（见图 6-9 和图 6-10）。

在图 6-9 中，我们用一个带标记的有向图说明了四个城市的 TSP 问题。**图**是结点和边的集合。**有向图**中的边是有方向的（边的末端是尾部，另外一端是头部）。如果一个图中的结点和边带有**标记**，我们就称之为**带标记**的图。在我们的例子中，有向图的结点对应于 TSP 问题中的城市，边对应于城市之间的路径，边上的标记对应于城市之间路径的代价。例如，从城市 0 到城市 1 的代价为 1，从城市 1 到城市 0 的代价为 5。

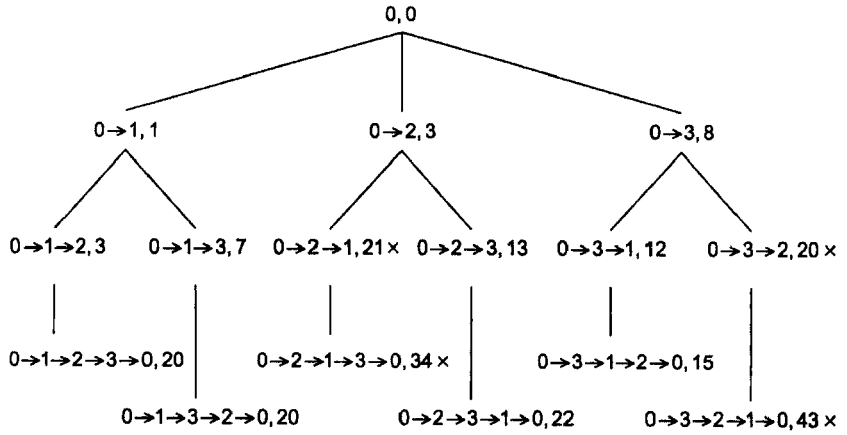


图 6-10　4 个城市 TSP 问题的搜索树

如果我们选取城市 0 作为旅行商的出发城市，因为我们还没有出发，所以初始的部分回路由结点 0 组成，代价为 0。因此，图 6-10 中代表部分回路的树只有结点 0，且代价为 0。从结点 0 出发，我们可以访问结点 1、结点 2 或者结点 3，代价分别为 1、3 和 8。在图 6-10 中，树的根有 3 个子结点。继续沿结点搜索，我们得到 6 个包含 3 个城市的部分回路，因为只有 4 个城市，所以
300 一旦选择了 3 个城市后，也就知道完整的回路了。

现在，要想找出最小代价的回路就需要搜索整个树。树搜索有很多方法，其中最常用的一种

就是**深度优先搜索**。在深度优先搜索的过程中，需要尽可能深地探测树的结构。一旦搜索到叶子结点或者搜索到不可能得到最小代价回路的结点，就需要回退到拥有未被访问子结点的“祖先”结点，并且开始尽可能深地探测其中的一个子结点。

在上面的例子中，从根结点开始，一直往左搜索直到某个叶子结点，标记分别为：

0→1→2→3→0，代价为 20

然后回退到标记为 0→1 的结点，因为它有未访问的子结点。继续分支搜索到叶子结点，标记为：

0→1→3→2→0，代价为 20

继续同样的过程，回退到根结点，转移到另一个分支，搜索标记为 0→2 的结点，访问它的子结点，标记为：

0→2→1，代价为 21

到达此结点后，就不需要再继续往下层搜索了，因为当前已经有比 21 代价还小的完整回路。回退到结点 0→2，并继续访问其未被访问的子结点，最终找出最小代价的回路是：

0→3→1→2→0，代价为 15

301

### 6.2.1　递归的深度优先搜索

深度优先搜索可以系统地访问树中每一个可能得到最小代价的结点，其最简单的形式就是使用递归（见程序 6-4）。程序 6-4 中第 8 ~ 13 行的 **for** 循环中，如果对城市的访问是按确定顺序，那么就比较容易实现递归。所以，我们假设城市的访问顺序是按城市的标记递增的，从城市 1 到城市 $n-1$。

递归的深度优先搜索使用以下几个全局变量：

- `n`：TSP 问题中城市的数量。
- `digraph`：代表输入有向图的数据结构。
- `hometown`：代表结点或者城市 0（即旅行商的家乡或者起点）的数据结构。
- `best_tour`：代表当前最佳回路的数据结构。

函数 `City_count` 用于测试部分回路 `tour`，看部分回路是否已经访问了 $n$ 个城市，如果是，就可以返回起点并完成一次旅行。然后调用 `Best_tour` 来检测该完全回路的代价是否比当前最佳回路的代价小。如果其代价更小，就调用 `Update_best_tour`，用其替代当前的最佳回路。注意，在第一次调用 `Depth_first_search` 之前，变量 `best_tour` 应该初始化为比任意可能的最佳回路代价大的代价。

如果部分回路 `tour` 还没有访问 $n$ 个城市，那么就需要继续分支搜索下去，换句话说，就是从部分回路的最后一个城市开始去继续访问其他未被访问的城市。要做到这点，需要做进一步的循环操作。函数 `Feasible` 用于检测结点是否已被访问过，如果没有被访问过，就检测访问它是否可能获得更小的回路代价。如果这个城市是可行的（即可能获得更小的回路代价），就把它加入到回路中，然后递归调用 `Depth_first_search`。当从 `Depth_first_search` 返回时，从回路中删除该城市，因为在随后的递归调用中的下一个回路不应该再包含该城市。

**程序 6-4　旅行商问题递归深度优先搜索的伪代码**

302

```
1  void Depth_first_search(tour_t tour) {
2     city_t city;
3
4     if (City_count(tour) == n) {
5        if (Best_tour(tour))
6           Update_best_tour(tour);
7     } else {
8        for each neighboring city
```

```
 9            if (Feasible(tour, city)) {
10               Add_city(tour, city);
11               Depth_first_search(tour);
12               Remove_last_city(tour, city);
13            }
14      }
15   }  /* Depth_first_search */
```

### 6.2.2 非递归的深度优先搜索

因为函数调用的开销很大，所以递归调用会运行得很慢。同时，在运行过程中只有当前树结点可以被访问，这样当我们想把树结点划分开来分配给各个线程或进程并行化树搜索时就会碰到问题。

非递归的深度优先搜索是可能实现的。基本思想是模拟递归实现，递归的函数调用可以通过把当前递归函数的状态压入运行时栈来实现。因此，在进入更深一层的树分支之前，把一些必要的数据压入自己的栈，可以通过这种方式来减少递归操作。在需要回溯时（可能是到达了叶子结点，也可能是搜索到一个不可能得到更好解的结点），就执行出栈操作。

**程序 6-5　旅行商问题的非递归深度优先搜索的伪代码**

```
 1  for (city = n-1; city >= 1; city--)
 2     Push(stack, city);
 3  while (!Empty(stack)) {
 4     city = Pop(stack);
 5     if (city == NO_CITY) // End of child list, back up
 6        Remove_last_city(curr_tour);
 7     else {
 8        Add_city(curr_tour, city);
 9        if (City_count(curr_tour) == n) {
10           if (Best_tour(curr_tour))
11              Update_best_tour(curr_tour);
12           Remove_last_city(curr_tour);
13        } else {
14           Push(stack, NO_CITY);
15           for (nbr = n-1; nbr >= 1; nbr--)
16              if (Feasible(curr_tour, nbr))
17                 Push(stack, nbr);
18        }
19     }  /* if Feasible */
20  }  /* while !Empty */
```

程序 6-5 实现了上述思路导出的非递归迭代深度优先搜索。在这个版本里，一条栈记录是一个城市，当一条记录出栈时，该城市就加入回路中。在递归版本里，程序重复递归调用直到已经访问了树中一条可行回路中的所有结点。到达这一结点后，栈中就不会再添加任何用于调用
303 Depth_first_search 的活动记录，然后程序返回至最开始 Depth_first_search 的调用点。在非递归迭代版本里，程序的主要控制结构是第 3 ~ 20 行的 **while** 循环，而循环终止的条件是栈为空。只要搜索仍然继续下去，就需要保证栈是非空的。在代码的前两行，我们把非起始点加入到栈中。需要注意的是，这个两行代码的循环按照降序访问城市，即从 $n-1 \sim 1$，这是由栈这个数据结构“后入先出”的特点决定的。将这个顺序反过来，这样就保证了在该版本中访问城市的顺序与递归版本中是一样的。

同时还注意到，第 5 行检查出栈城市是否是常量 NO_CITY。该常量用于表示树中某结点的所有子结点都已经被访问过了。如果不用这个常量，就无法决定什么时候执行回溯操作。因此，在把某结点的所有子结点压入栈前（第 15 ~ 17 行），需要把 NO_CITY 入栈。

对上述非递归迭代版本还可以有另一个替代版本：用部分回路作为栈中的记录（见程序 6-6）。

这样的代码更接近于递归函数的形式。然而，这个版本的执行速度会更慢，因为入栈函数在把回路压入栈中前需要创建该回路的副本。为了强调这一点，程序中调用的是 Push_copy 函数。（如果只是简单地将指向当前回路的指针压入栈，会出现什么问题?）运行这个版本的程序需要更多内存，这不会是大问题。但是，给新回路分配内存空间和复制现有回路是很耗时的。在某种程度上，可以考虑把已经释放的回路保存在自己的数据结构中，以便在函数 Push_copy 利用已经释放的回路时不必调用 malloc，从而降低上述操作的开销。

**程序 6-6　旅行商问题的非递归深度优先搜索的另一种伪代码**

```
1  Push_copy(stack, tour);  // Tour that visits only the hometown
2  while (!Empty(stack)) {
3     curr_tour = Pop(stack);
4     if (City_count(curr_tour) == n) {
5        if (Best_tour(curr_tour))
6           Update_best_tour(curr_tour);
7     } else {
8        for (nbr = n-1; nbr >= 1; nbr--)
9           if (Feasible(curr_tour, nbr)) {
10             Add_city(curr_tour, nbr);
11             Push_copy(stack, curr_tour);
12             Remove_last_city(curr_tour);
13          }
14    }
15    Free_tour(curr_tour);
16 }
```

另一方面，该版本有一个特点，即栈或多或少独立于其他数据结构。因为整个回路都存储起来，所以多个线程/进程就可以方便地各自存取回路。如果很小心地进行设计，就不会破坏程序 304
的正确性。在之前的非递归迭代版本里，一个栈记录就是一个城市，它没有提供足够的现场信息告知我们当前搜索过程所处的阶段。

### 6.2.3　串行实现所用的数据结构

程序中基本的数据结构包括回路、有向图和栈（在迭代实现中要使用到）。回路和栈一般用链表结构表示。在经常处理的 TSP 问题中，城市的数目一般都比较小（小于 100），采用链表来表示回路没有优势，所以我们用数组存储 $n+1$ 个城市。因为我们需要重复获取部分回路中城市的数量和部分回路的代价，所以我们用结构体替代简单的数组来表示回路，该结构体的成员包括：存储城市的数组、城市的数量和部分回路的代价。

为了提高代码的可读性和性能，用预处理器宏存取结构体的成员变量。然而，因为宏对于程序调试来说是一个噩梦，所以在初始开发阶段使用“存取器”函数是个不错的主意。当“存取器”函数可以正常工作后，再用宏替代它。例如，我们可能开始用的函数是：

```
/* Find the ith city on the partial tour */
int Tour_city(tour_t tour, int i) {
   return tour->cities[i];
}  /* Tour_city */
```

当程序正常工作后，我们可以用下面的宏来替代：

```
/* Find the ith city on the partial tour */
#define Tour_city(tour, i) (tour->cities[i])
```

在第一个非递归的迭代版本中，栈只是一组城市或者一组整数值（int）。而且，任意时刻在栈中不会有超过 $n^2/2$ 个记录，同时 $n$ 又比较小，所以可以像回路的数据结构那样，采用数组结构来表示，并把栈中元素的数量也存储起来。因此，Push 可以用下面的代码实现：

```
void Push(my_stack_t stack, int city) {
   int loc = stack->list_sz;
   stack->list[loc] = city;
   stack->list_sz++;
}  /* Push */
```

在第二种非递归迭代版本里，栈中的元素是回路，我们也可以用数组来表示回路。入栈函数类似于下面的代码：

```
void Push_copy(my_stack_t stack, tour_t tour) {
   int loc = stack->list_sz;
   tour_t tmp = Alloc_tour();
   Copy_tour(tour, tmp);
   stack->list[loc] = tmp;
   stack->list_sz++;
}  /* Push */
```

305 而且，对栈中元素的访问也可以用宏来实现。

有向图的数据结构可以有多种实现形式。当有向图的边数比较少时，链表是较好的实现形式。然而，在我们的设定里，两个不同结点之间相互连接的边是有向的，结点 $i$ 到结点 $j$ 的边和结点 $j$ 到结点 $i$ 的边是不同的，有着各自的权值。所以需要存储每条有向边的权值，采用**邻接矩阵**比链表结构更合适。也就是 $n \times n$ 的矩阵，结点 $i$ 到结点 $j$ 的边的权值就是矩阵第 $i$ 行第 $j$ 列的元素值。我们可以直接存取这个权值，而不需要遍历链表。矩阵对角线上的元素（第 $i$ 行和第 $i$ 列的元素）没有使用，因此设置值为 0。

### 6.2.4 串行实现的性能

上述三种串行实现的运行时间如表 6-7 所示。输入的有向图含有 15 个结点（包括起始点），三种算法访问了大约 9 500 万个树结点。第一个迭代版本比递归版本要快约 5%，第二个迭代版本比递归版本要慢约 8%。正如所期望的那样，第一个迭代版本减少了某些重复函数调用所带来的开销，同时第二个迭代版本由于重复地复制回路数据结构而运行得更慢。然而，第二个迭代版本更易于实现并行化，因此将采用它作为树形搜索并行化版本的基础。

表 6-7 树形搜索的三种串行化实现的运行时间（单位：秒）

| 递归实现 | 第一个迭代版本 | 第二个迭代版本 |
|---|---|---|
| 30.5 | 29.2 | 32.9 |

### 6.2.5 树形搜索的并行化

先看看树形搜索的并行化方案。树的结构提示我们按照树结点来划分任务。如果这样做，任务将随着树的边往下传递：父结点传递一个新的部分回路给子结点，但是，除非终止搜索，否则子结点不会直接与父结点通信。

我们还需要考虑对最佳回路的更新和使用。每个任务都通过测试最佳回路来判定当前部分回路是否可行或者当前完整回路是否有更低的代价。如果一个任务（到达叶子结点时）判定它的回路代价更低，那么就会更新最佳回路。尽管所有实际的计算可以通过分配给树结点的任务来实
306 现，但还要注意到，最佳回路数据结构会增加额外的、非显式通过树边表示的通信。因此，最好增加一个任务来专门处理最佳回路。该任务给每个树结点中的任务发送数据，同时接收来自某些叶子结点的数据。该任务在共享内存系统中较容易实现，但在分布式内存系统中实现起来却不是太方便。

在分配和映射任务时，一个自然而然的做法是将一个子树分配给一个线程/进程，由每个线程/进程执行子树的所有任务。例如，有 3 个线程或者进程（如图 6-10 所示），可以把0→1 为根

的子树映射给线程/进程 0，以 0→2 为根的子树映射给线程/进程 1，以 0→3 为根的子树映射给线程/进程 2。

**映射操作的细节**

如何将子树分配给线程/进程有很多种算法。例如，一个线程/进程可以运行最后一个版本的串行化深度优先搜索，直到所有线程/进程的栈里都存有一条部分回路。这样就可以分配一条回路给一个线程/进程。深度优先搜索存在的问题是：根结点越高的子树需要做的工作越多，所以，为了获得较好的负载均衡，可以使用类似于**广度优先搜索**的方法来找出子树。

正如名字所暗示的，广度优先搜索在搜索更深层次前先尽可能广地访问结点。因此，如果实现一次广度优先搜索，直到搜索到树的某层，其结点数至少有 thread_count 或者 comm_sz，就可以在这一层划分任务。习题 6.18 有更详细的实现。

**最佳回路的数据结构**

在共享内存系统里，最佳回路的数据结构可以共享。在这种设定下，Feasible 函数可以简单地测试这个数据结构。然而，对这个数据结构的更新会导致竞争，需要对锁进行排序来避免这类错误。在实现并行版本时，我们会对此问题给出更详细的讨论。

在分布式内存系统里，对最佳回路的处理有多种选择。最简单的一种是让各个进程独立工作，直到它们分别完成了对各自子树的搜索。在这种设定里，每个进程需要存储自己的最佳回路。这个本地的局部最佳回路方案会由进程在 Feasible 函数里调用，并且在每次调用 Update_best_tour 时得到更新。当所有的进程都完成了各自的搜索后，它们将执行一个全局的归约操作来找出全局最少代价的回路，即最佳回路。

这种方法比较简单，但存在一个问题，即有可能某个进程花费了所有的时间去搜索一个不可能得到全局最佳回路的子树。因此，应该让当前全局最佳回路对于所有的进程都可见。我们会在 307 讨论 MPI 实现时给出更详细的介绍。

**动态映射任务**

我们需要考虑的第二个问题是负载不均衡。尽管使用广度优先搜索可以确保所有划分的子树有近似的结点数，但是却不能保证它们有同样的工作负载。完全有可能存在这样的情况：分配给某个线程/进程子树的回路代价很高，不需要对子树进行很深度的搜索。目前来看，在静态任务映射方案中，该线程/进程将简单地等待，直到其他线程/进程完成任务为止。

对于上述问题，一个可替代的方案是**动态**任务映射。在动态方案中，当一个线程/进程完成分配给它的任务时，就可以从另一个线程/进程那里分配获得额外的任务。在我们最终实现的串行深度优先搜索里，栈的每个记录是一个部分回路。使用这种数据结构，某个线程/进程就可以通过划分自己栈中的内容将任务分配给另一个线程/进程。这样做可能会出现程序正确性的问题，因为如果把一个线程栈里的部分工作分配给另一个线程，那么树结点被访问的顺序有可能被改变。

然而，我们打算这样做：当分配不同的子树给不同的线程/进程时，树结点被访问的顺序就不再是串行深度优先搜索的顺序了。实际上，只要能保证：访问“祖先”一定在“后代”之前，就不需要保证访问某个结点必须在另一个其他结点之前。这一点其实并不是问题，因为一个部分回路在它所有的“祖先”结点都被访问前是不会被压入栈中的。例如，在图 6-10 中，当含有回路 0→2 的结点是当前动态结点时，含有回路 0→2→1 的结点会被压入栈中，因此这两个结点不会同时在栈中。类似地，0→2 的父结点，也就是树的根结点，在0→2 被访问后，也不会再在栈中。

另一种动态负载均衡方案（至少在共享内存的情况下）使用共享栈。然而，我们不可以简单

地去掉那些本地栈。如果一个线程需要在每次入栈和出栈操作时存取共享栈，那么会产生非常多的竞争问题，并行程序的性能也可能会比串行程序更差。实际上，这也是在用互斥量/锁并行化 $n$ 体问题时碰到的问题。如果每次调用 Push 和 Pop 操作都要访问临界区一次，程序就会近似于完全暂停。因此，我们会为每个线程保留本地栈，只在少数时候访问共享栈。我们不会实现这种替
308 代方案。编程作业 6.7 对此有更详细的讨论。

### 6.2.6 采用 Pthreads 实现的静态并行化树搜索

在静态并行化中，单个线程使用深度优先搜索生成足够的部分回路，这使得每个线程至少分配到一条部分回路。然后每个线程对分配给它的回路进行迭代搜索。每个线程可以采用如程序 6-7 所示的伪代码。值得注意的是，对于绝大多数的函数调用（例如 Best_tour、Feasible、Add_city），需要存取代表有向图的邻接矩阵，所以所有线程都需要存取有向图。这些只是读操作，所以不会导致线程间的竞争冲突。

**程序 6-7 TSP 问题的静态并行化 Pthreads 实现的伪代码**

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n-1; city >= 1; city--)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

这里的伪代码与第二个迭代串行实现里的伪代码主要有个四个不同的方面：

- 采用 my_stack 来替代 stack；因为每个线程都有自己的私有栈，所以使用 my_stack 作为栈的标识符，而不采用 stack。
- 栈的初始化。
- Best_tour 函数的实现。
- Update_best_tour 函数的实现。

在串行实现中，栈的初始化把只包含起始点的部分回路压入栈中。在并行化版本里，我们需要生成至少 thread_count 数量的部分回路，并将这些回路分配给线程组中的每个线程。正如我们之前讨论的那样，让某个线程进行广度优先搜索，直到到达至少有 thread_count 个子树
309 的层次，这样可以生成至少 thread_count 条回路（需要注意的是：这意味着线程的数量应该小于 $(n-1)!$）。然后，所有的线程就可以采用块划分来分配这些回路，并把它们分别压入到各自的私有栈中。习题 6.18 对此有详细的探讨。

为了实现 Best_tour 函数，每个线程都需要把自己当前回路的代价与全局最佳回路的代价进行对比。因为多个线程可能同时访问全局最佳回路的代价，这就可能存在竞争访问。然而，Best_tour 函数只是读取全局最佳回路的代价，所以这些读取操作之间不会存在冲突。如果一个线程正在更新全局最佳回路的代价，此时另一个线程又正好检查最佳回路代价的值，那么就会既可能读取到的是旧值，也可能是更新过的值。当然，我们希望读取到的是更新值，但是如果不采用一

些复杂的锁策略就不能保证读取到的是更新过的值。例如，想要执行 Best_tour 或者 Update_best_tour 的线程可能需要在一个互斥量上等待。这样可以保证没有线程可以在另一个线程只是检查值的时候去更新值，但这种做法也会产生一个新问题：边际效应，即一次只能有一个线程可以检查最佳回路的代价。还可以采用读写锁来改进这个方法，但这种方法也有副作用——调用 Best_tour 的线程，会由于另一个线程更新最佳回路的代价而阻塞。原则上，这听起来好像不太坏，但实际上使用读写锁会相当慢。因此，可能是虽然得到过时的数据但无冲突的方案效果会更好，因为在线程下一次调用 Best_tour 时它就会得到更新过的最佳回路的代价。

另一方面，调用 Update_best_tour 会在线程之间同时写时出现写冲突。为了避免这个问题，可以用互斥量来保护 Update_best_tour 函数。不过这还不够，在某个线程完成 Best_tour 中对最佳回路代价的检查与它获得 Update_best_tour 中的锁之间，有可能另一个线程先取得锁并更新最佳回路代价，而这个代价可能会小于第一个线程在 Best_tour 中发现的最佳回路代价。因此，Update_best_tour 正确的伪代码应该类似于：

```
pthread_mutex_lock(best_tour_mutex);
/* We've already checked Best_tour, but we need to check it
    again */
if (Best_tour(tour))
   Replace old best tour with tour;
pthread_mutex_unlock(best_tour_mutex).
```

这样看起来似乎有些浪费，但是如果最佳代价的更新不是很频繁的话，那么绝大多数的 Best_tour 会返回 false，而且几乎不需要做“二次”调用。

### 6.2.7 采用 Pthreads 实现的动态并行化树搜索

如果将子树初始化分配给各个线程的工作没有做好，那么静态并行化策略也无法对工作进行重新分配。那些分配到“小”子树的线程就会很早完成工作，同时那些分配到大子树的线程却会继续工作。这其实不难想象：某个线程所获得的初始回路中边的代价较小，而其他线程 310
的初始回路中边的代价较大。为了解决这个问题，可以尝试在计算的过程中动态重新分配工作。

为了做到这一点，我们用更复杂的代码去替代 **while** 循环的！Empty(my_stack)控制执行条件。其基本原理是：当一个线程做完工作以后，即！Empty(my_stack)变成 false 时，线程不会立即离开 **while** 循环，而是等待看是否有另外一个线程可以提供更多的工作。另一方面，如果一个线程还有工作可以做，同时发现有其他线程没有工作可以做，并且它的栈里有至少两个回路时，该线程可以“分离”它自己的栈，把没做完的工作提供给其他线程去完成。

Pthreads 的条件变量提供了一个很简单的方法来实现上述想法。当一个线程做完工作后，它调用 pthread_cond_wait，然后进入睡眠。当一个有工作可做的线程发现至少有一个线程正在等待分配任务时，在分离自己的栈后，该线程调用 pthread_cond_signal 唤醒处于睡眠态的线程。被唤醒的线程获得被分离的栈中一半的任务并返回工作状态。

这个思想可以扩展用于处理线程的终止。如果保持一定数量的线程处于 pthread_cond_wait 状态，当某个栈为空的线程发现已经有 thread_count −1 个线程处于等待分配任务的状态时，它就可以调用 pthread_cond_broadcast，让所有睡眠的线程都醒来，这时就会发现所有的线程都完成了工作，即可以退出执行了。

#### 终止状态

我们可以使用如程序 6-8 所示 Terminated 函数的伪代码来代替 **while** 循环中的 Empty 来实现树的搜索。

**程序 6-8** Terminated 函数的 Pthreads 伪代码

```
1  if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
2        new_stack == NULL) {
3     lock term_mutex;
4     if (threads_in_cond_wait > 0 && new_stack == NULL) {
5        Split my_stack creating new_stack;
6        pthread_cond_signal(&term_cond_var);
7     }
8     unlock term_mutex;
9     return 0;  /* Terminated = false; don't quit */
10 } else if (!Empty(my_stack)) /* Keep working */
11    return 0;  /* Terminated = false; don't quit */
12 } else { /* My stack is empty */
13    lock term_mutex;
14    if (threads_in_cond_wait == thread_count-1)
15                                    /* Last thread running */
16       threads_in_cond_wait++;
17       pthread_cond_broadcast(&term_cond_var);
18       unlock term_mutex;
19       return 1;  /* Terminated = true; quit */
20    } else { /* Other threads still working, wait for work */
21       threads_in_cond_wait++;
22       while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
23       /* We've been awakened */
24       if (threads_in_cond_wait < thread_count) { /* We got work */
25          my_stack = new_stack;
26          new_stack = NULL;
27          threads_in_cond_wait--;
28          unlock term_mutex;
29          return 0;  /* Terminated = false */
30       } else {  /* All threads done */
31          unlock term_mutex;
32          return 1;  /* Terminated = true; quit */
33       }
34    }  /* else wait for work */
35 }  /* else my_stack is empty */
```

这里，有几个细节需要更进一步地探讨。注意：一个线程在分离自己的栈之前需要执行的代码相当复杂。在线程的第 1 ~ 2 行代码中：

- 检查栈中是否拥有至少两个回路。
- 检查是否有线程等待。
- 检查变量 new_stack 是否为 NULL。

检查线程是否有足够工作的原因很清楚：如果栈里的记录少于两条，“分离”栈的操作就会什么都不做，或者导致两个线程之间的角色互换：活动线程被某一个等待线程所替代。

而且很清楚的是：如果没有线程等待任务，分离栈的操作就是没有必要的。最后，如果某些线程已经分离了各自的栈，但等待线程却没有检查新栈是否为空，也就是说，此时 new_stack != NULL，那么分离一个栈并对现存的新栈进行重写的操作就会导致出错。要注意的是，某个线程在检查 new_stack 后，也就是复制 new_stack 到自己私有的 my_stack 后，必须把 new_stack 设置为 NULL，这样做非常有必要。

如果以上三个条件都满足，那么就可以尝试分离线程自己的栈。我们需要获取用于保护控制终止条件（threads_in_cond_wait、new_stack 和条件变量）的互斥量。然而，条件：

```
threads_in_cond_wait > 0 && new_stack == NULL
```

311 可能会在开始等待互斥量与获取到互斥量之间的这段时间内改变，所以就像 Update_best_tour 一样，需要确保在获取互斥量后条件依然满足（第 4 行）。一旦证实了这个条件保持未变，那么就可以分离栈，唤醒某个等待线程，解锁互斥量，然后返回去工作。

如果第 1 行和第 2 行的检查为 false，那么就去检查一下是否有工作需要做，即栈是否为空。如果是，就返回去工作；如果不是，通过等待、获取终止状态的互斥量（第 13 行），启动终止线程的序列。一旦取得了互斥量，就有两种可能性： 312

- 这是最后一个进入终止序列的线程，也就是 threads_in_cond_wait == thread_count -1。
- 有其他线程仍然在工作。

第一种情况下，如果知道其他所有线程已经做完了工作，同时该线程也完成了任务，那么树搜索就可以终止了。因此，可以调用 pthread_cond_broadcast 通知所有其他线程，然后返回 true。在执行广播前，即使广播是要告诉所有的线程从条件等待返回，变量 thread_in_cond_wait 也需要加 1。理由是 threads_in_cond_wait 有双重作用：当它小于 thead_count 时，它代表等待线程的数量；当它等于 thread_count 时，意味着所有的线程都做完了工作，应该退出执行了。

第二种情况下，可以调用 pthead_cond_wait（第 22 行），然后等待被唤醒。一个线程有可能被除了 pthread_cond_signal 和 pthread_cond_broadcast 以外的操作唤醒。因此，把调用 pthread_cond_wait 放在 **while** 循环中，这样，如果有其他事件（返回的值非 0）唤醒线程时就会立即调用 pthread_cond_wait。

一旦被唤醒，就需要考虑两种情况：

- threads_in_cond_wait < thread_count。
- thread_in_cond_wait = = thread_count。

在第一种情况中，我们已知一些线程已经分离了自己的栈，生成了更多的工作。因此可以将最近生成的栈复制到自己的私有栈中，并设置变量 new_stack 为 NULL，并将 threads_in_cond_wait 的值减 1（第 25 ~ 27 行）。考虑到，当一个线程从条件等待返回时，它获取了条件变量相关的互斥量，因此在返回前，需要释放互斥量（第 28 行）。在第二种情况中，没有可以做的工作，所以释放互斥量并返回 true。

在实际代码中，我们发现把条件变量放入一个独立的结构体里非常方便。因此，可以如下定义：

```
typedef struct {
   my_stack_t new_stack;
   int threads_in_cond_wait;
   pthread_cond_t term_cond_var;
   pthread_mutex_t term_mutex;
} term_struct;
typedef term_struct* term_t;

term_t term;  // global variable
```

同时，定义一组函数，一个用于初始化 term 变量，一个用于释放变量和它的成员。

在讨论分离栈的函数前，需要注意一个正在工作的线程在它可以分离栈前有可能花费很多时间等待 term_mutex。其他线程此时可能正在分离它们的栈，也有可能正在进行条件等待。如果 313
担心这个问题，Pthreads 提供了一个非阻塞的替代方案（替代 pthread_mutex_lock），称为 pthread_mutex_trylock：

```
int pthread_mutex_trylock(
         pthread_mutex_t*    mutex_p    /* in/out */);
```

这个函数尝试获取 mutex_p。但是，如果该互斥量已经被锁上了，那么它不会等待而是直接返回。如果调用线程成功获取了互斥量，那么返回值为 0；否则，返回非 0。作为分离栈操作前等待互斥量的替代方法，线程可以调用 pthread_mutex_trylock。如果成功获取了 term_mu-

tex，就可以如之前的那样继续执行；如果没有，就会直接返回。

**分离栈**

因为我们的目标是平衡线程之间的负载，所以我们应尽量确保新栈中任务的数量与留在原栈中任务的数量一致。我们无法提前知道从某个部分回路开始搜索子树的实际工作量有多少，所以我们不可能保证对任务进行等量划分。但是可以采用之前初始化分配子树的方法：从两个拥有同样数目城市的部分回路开始出发的子树有类似的结构。因为两个拥有同样数目城市的部分回路很有可能在任务量上也是接近一致的，所以可以基于边的数目来划分栈。边数量最少的回路留在原始栈里，数量接近最少边数的部分回路分配给新栈，下一个次少边数的部分回路保留在原始栈中，然后以次类推地分配下去。

上述思路是比较容易实现的，因为栈中保留的回路的边数是依次递增的。也就是说，从栈底到栈顶，每条栈记录保存的回路边数是递增的。因为当我们把一个新的有 $k$ 条边的部分回路压入栈时，紧邻它之前的记录就有 $k$ 或者 $k-1$ 条边。可以从栈的底部开始划分，并依次轮流把回路压入旧栈和新栈中，所以回路 0 会留在旧栈中，回路 1 会分配到新栈中，回路 2 会留在旧栈中，依次这样分配下去。如果栈是用一个回路数组来实现的，那么就需要对旧栈进行“压缩”，以消除因为移出部分回路所产生的空白。如果栈是用链表来实现的，就不必要执行压缩了。

可以进一步考虑到，拥有很多城市的部分回路不会导致很重的工作负载，因为以这些回路为根的子树都很小。因此可以增加一个“截止大小”（cutoff size），如果城市的数量不小于该截止
314 大小，就不重新分配任务。在共享内存并采用基于数组的栈的设置下，因为回路（是一个指针）会复制到新栈中或者分配到旧栈的新位置，所以分离栈时重新分配回路不会增加分离栈的开销。将在编程作业 6.6 中讨论这种方法。

### 6.2.8 Pthreads 树搜索程序的评估

表 6-8 显示了两个有 15 个城市的 Pthreads 程序的性能。“串行”列给出了第二种迭代方案（把每一个新的回路压入栈中）的运行时间。作为参考，表中的第一个问题采用类似于表 6-7 中的测试方法，Pthreads 和串行实现都是在同一个系统下测试。运行时间以秒为单位。在动态划分版本的程序运行时间旁边的括号里，给出的是栈被分离的总次数。

**表 6-8 树搜索 Pthreads 程序的运行时间（单位：秒）**

| 线程 | 第一个问题 | | | | 第二个问题 | | | |
|---|---|---|---|---|---|---|---|---|
| | 串行 | 静态 | 动态 | | 串行 | 静态 | 动态 | |
| 1 | 32.9 | 32.7 | 34.7 | (0) | 26.0 | 25.8 | 27.5 | (0) |
| 2 | | 27.9 | 28.9 | (7) | | 25.8 | 19.2 | (6) |
| 4 | | 25.7 | 25.9 | (47) | | 25.8 | 9.3 | (49) |
| 8 | | 23.8 | 22.4 | (180) | | 24.0 | 5.7 | (256) |

从运行时间可以看出，不同的问题会有不一样的结果。例如，使用静态划分的程序通常在第一个问题上表现得比动态划分好。然而，在第二个问题上，静态划分程序的性能与线程数没有什么关系，而动态划分程序却有更好的性能。一般来说，动态划分程序的可扩展性比静态划分程序要好。

随着线程数量的增加，私有栈的大小会变小，因此线程会更容易早些完成工作。可以预测的是，当某些线程处于等待状态时，其他线程分离自己的栈，所以随着线程数量的增加，分离栈的操作会增加。这两个问题都证实了这个预测。

需要注意的是，如果输入问题有多于一个可行解，也就是说，不同的回路有同样大小的代价，这两种程序的结果都是不确定的。在静态程序里，最佳回路的顺序依赖于线程执行的速度，而这个

顺序决定哪些树结点被访问。在动态程序里，结果同样是不确定的，因为不同的运行可能会导致线程划分旧栈和接收新栈的地方是不同的。这会导致运行时间，尤其是动态运行时间变化很大。 315

### 6.2.9 采用 OpenMp 实现的并行化树搜索程序

这个问题涉及采用 OpenMp 实现静态和动态并行化树搜索程序，有些类似于 Pthreads 采用的方法。

在静态并行化方面，采用 OpenMp 和采用 Pthreads 几乎没有什么不同。但是，也有一些要点需要注意：

- 在 Pthreads 中的一个独立线程执行某部分代码时，测试：

  ```
  if (my_rank == whatever)
  ```

  可以被 OpenMp 的指令：

  ```
  #  pragma omp single
  ```

  替代。

  这样做能确保接下来的结构化代码块由线程组中的一个线程执行，而组内其他线程会等待直到该线程执行结束（在代码块的最后设置一个隐式的路障）。

  当 whatever 等于 0 时（正如 Pthreads 程序里测试的那样），可以用 OpenMp 的指令：

  ```
  #  pragma omp master
  ```

  替代。

  这样能确保由线程 0 来执行接下来的结构化代码块。然而，master 指令不会在代码块的最后设置隐式的路障，所以很有必要在代码块的最后加入一条 barrier 指令。
- Pthreads 里用于保护最佳回路的互斥量可以被一条 critical 指令所替代，这个操作可以发生在 Update_best_tour 函数内部或者调用 Update_best_tour 函数前。这是在初始化分配回路后唯一个可能存在竞争的地方，所以简单的 critical 指令不会导致某个线程一直阻塞。

动态负载均衡的 Pthreads 实现严重依赖于 Pthreads 的条件变量，而 OpenMp 没有提供这样一个类似的实现。剩余其他的 Pthreads 代码可以很容易地转换为 OpenMp 代码。实际上，OpenMp 甚至还提供了非阻塞版本的 omp_set_lock。提供的锁对象类型是 omp_lock_t，下面分别是获取和释放锁的函数：

```
void omp_set_lock(omp_lock_t*    lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*  lock_p   /* in/out */);
```

它还提供了函数：

```
int omp_test_lock(omp_lock_t*    lock_p   /* in/out */);
```

这个函数类似于 pthread_mutex_trylock；它尝试去获取锁 * lock_p，如果成功获取，就返回非 0 值。如果该锁已经被其他线程获取了，它就立即返回 0 值。 316

如果查看一下程序 6-8 中 Pthreads Terminated 函数的伪代码，就会发现为了把 Pthreads 版本修改为 OpenMp 形式，需要模拟 Pthreads 的函数调用：

```
pthread_cond_signal(&term_cond_var);
pthread_cond_broadcast(&term_cond_var);
pthread_cond_wait(&term_cond_var, &term_mutex);
```

这三个函数调用分别在程序的第 6 行、第 17 行和第 22 行。

回忆一下，线程通过以下调用进入条件等待：

```
pthread_cond_wait(&term_cond_var, &term_mutex);
```

这样的调用会等待两种事件：

- 另一个线程已经分离了自己的栈，并为正在等待的线程创建了任务。
- 所有的线程都完成了工作。

在 OpenMp 里，模拟条件等待最简单的一种做法就是使用忙等待。因为等待线程需要测试两个条件，可以在忙等待循环里使用两个不同的变量。

```
/* Global variables */
int awakened_thread = -1;
int work_remains = 1;  /* true */
. . .
while (awakened_thread != my_rank && work_remains);
```

这两个变量的初始化是非常重要的：如果 awakened_thread 初始化为某些线程的线程号，那么这个线程就会立即从 **while** 循环中退出，但这时可能还有工作可以做。类似地，如果 work_remains 初始化为 0，那么所有的线程都会立即从 **while** 循环中退出并退出执行。

正如之前分析过的，Pthreads 中的某个线程进入条件等待时，它会释放与条件变量相关的互斥量，以使得另一个线程可以进入条件等待或者发信号唤醒等待线程。因此，可以在开始 **while** 循环前释放在 Terminated 函数中使用的锁。

当一个线程从 Pthreads 条件等待返回时，它会释放与条件变量相关的互斥量。这是非常重要的，因为如果已经唤醒的线程接收了工作，那么它就需要获取存放在新栈的共享数据结构。因此，完整地模拟条件等待应该类似于：

```
/* Global vars */
int awakened_thread = -1;
work_remains = 1;  /* true */
. . .
omp_unset_lock(&term_lock);
while (awakened_thread != my_rank && work_remains);
omp_set_lock(&term_lock);
```

317 如果回想一下 4.5 节的忙等待和习题 4.3 的讨论，就知道编译器可能会对忙等待循环的代码进行重新排序。但编译器不应该对 omp_set_lock 和 omp_upset_lock 进行重新排序。然而，对变量的更新可以会重新排序，所以如果使用编译器优化，就需要对这两个函数使用 volatile 关键字进行声明。

对条件广播（condition broadcast）的模拟是很直接的：当一个线程确定不再有工作需要执行时（程序 6-8 中的第 14 行），条件广播（第 17 行）可以用下面的表达式替代：

```
work_remains = 0;  /* Assign false to work_remains */
```

被“唤醒”的线程可以测试它们是否是由某些线程设置 work_remains 为 false 所唤醒的，如果是这样，就从 Terminated 返回，返回值为 true。

对条件信号量（condition signal）的模拟需要多做一点工作。已经分离了栈的线程需要选择一个正处于睡眠状态的线程，并把 awakened_thread 变量赋值为被选中的被唤醒线程的线程号。因此，至少要维持一组处于睡眠状态线程的线程号的列表。解决这个问题的一个简单做法是，使用一个线程号共享队列。当某个线程完成任务后，就在进入忙等待循环前把它的线程号加入到队列中。当一个线程分离栈时，它就可以通过出队列操作选择一个睡眠线程，并把它唤醒。

```
got_lock = omp_test_lock(&term_lock);
if (got_lock != 0) {
   if (waiting_threads > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      awakened_thread = Dequeue(term_queue);
   }
   omp_unset_lock(&term_lock);
}
```

在从调用的 Terminated 函数返回前，被唤醒的线程需要重置 awakened_thread 为 -1。

其他线程在被唤醒的线程重新取得锁前被唤醒是没有什么危害的。只要 new_stack 是非 NULL（空），就不会有线程尝试分离栈，因此也不会有线程唤醒另外一个线程。所以，如果一个线程在被唤醒的线程重新获取锁前调用 Terminated，那么如果它们的栈是非空的，它们就会返回，或者如果它们的栈为空，它们就会进入等待。

### 6.2.10　OpenMp 实现的性能

表 6-9 是有 15 个城市的旅行商问题的两种 OpenMP 实现的运行时间，该问题与 Pthreads 实现 318
和串行测试里的问题是一样的，并且运行在同一个系统里。为了方便对比，我们也标出了 Pthreads 实现的运行时间。运行时间是以秒为单位，括号内的数字是动态实现里栈被划分的总次数。

表 6-9　树搜索的 OpenMP 和 Pthread 实现的性能（单位：秒）

| 线程 | 第一个问题 | | | | | | 第二个问题 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 静态 | | 动态 | | | | 静态 | | 动态 | | | |
| | OMP | Pth | OMP | | Pth | | OMP | Pth | OMP | | Pth | |
| 1 | 32.5 | 32.7 | 33.7 | (0) | 34.7 | (0) | 25.6 | 25.8 | 26.6 | (0) | 27.5 | (0) |
| 2 | 27.7 | 27.9 | 28.0 | (6) | 28.9 | (7) | 25.6 | 25.8 | 18.8 | (9) | 19.2 | (6) |
| 4 | 25.4 | 25.7 | 33.1 | (75) | 25.9 | (47) | 25.6 | 25.8 | 9.8 | (52) | 9.3 | (49) |
| 8 | 28.0 | 23.8 | 19.2 | (134) | 22.4 | (180) | 23.8 | 24.0 | 6.3 | (163) | 5.7 | (256) |

在大部分情况下，OpenMp 实现可以与 Pthreads 实现的性能相当。这并不令人惊奇，因为系统运行在一个八核平台上，忙等待并不会降低系统的总体性能，除非使用超过核数量的线程。

对第一个问题来说，有两个比较特别的例外。采用 8 个线程的静态 OpenMp 实现的性能比 Pthreads 实现的性能要差，采用 4 个线程的动态 OpenMP 实现比 Pthreads 实现的性能要差。这可能是由于程序的不确定性导致的，不过我们还需要更详细的资料来确定原因。

### 6.2.11　采用 MPI 和静态划分来实现树搜索

采用 Pthreads 和 OpenMP 的静态并行化树搜索中绝大多数的代码，是直接借鉴第二个串行化迭代树搜索的实现。实际上，唯一的不同之处是，在线程开始时对树的初始划分和 Update_best_tour 函数。因此我们希望 MPI 实现也只需要对这部分代码进行少量修改，实际上也正是如此。

MPI 程序中一个常见的问题是需要分配输入的数据和收集结果。为了构建一条完整的回路，进程需要为每一个结点选择一条进入该结点和离开该结点的边。因此，对每一个加入回路的城市，回路都需要邻接矩阵中该城市对应的一行和一列的数据，因此如果每个进程都可以存取整个邻接矩阵就会十分方便。同时邻接矩阵所占用的存储空间相对较小，即使我们有 100 个城市，矩阵也不可能需要超过 80 000 字节的存储空间，所以只要进程 0 读取矩阵并将其传送给其他进程就 319
可以了。

一旦各个进程有了邻接矩阵的副本，大多数的树搜索任务都可以像 Pthreads 和 OpenMP 的实

现那样进行。其中最主要的不同在于：

- 划分树。
- 测试和更新最佳回路。
- 在搜索终止后，保证进程0有一个最佳回路的副本用于输出。

我们将在下面依次讨论上面的问题。

**划分树**

在Pthreads和OpenMP的实现里，线程0使用广度优先搜索来搜索树，直至部分回路的数量至少为thread_count才停止。然后，每个线程选取它应该得到的初始部分回路，并把这些回路压入自己的私有栈中。很明显，MPI的进程0也可以生成一组数目为comm_sz的部分回路。然而，因为不是共享内存系统，所以需要将各条初始部分回路发送给合适的进程。在此可以通过一个发送循环来做这个工作，而分发这些初始部分回路的操作类似于MPI_Scatter。实际上，不能使用MPI_Scatter的唯一原因是，初始部分回路的数目无法被comm_sz所整除。如果这个问题发生，进程0就无法发送相同数量的回路给每个进程，但MPI_Scatter需要发送相同数量的对象给每个进程。

幸运的是，存在一个MPI_Scatter的变体，MPI_Scatterv，它可以用来发送不同数量的对象给不同的进程。首先，回顾一下MPI_Scatter的语法：

```
MPI_Scatter
int MPI_Scatter(
      void          sendbuf     /* in  */,
      int           sendcount   /* in  */,
      MPI_Datatype  sendtype    /* in  */,
      void*         recvbuf     /* out */,
      int           recvcount   /* in  */,
      MPI_Datatype  recvtype    /* in  */,
      int           root        /* in  */,
      MPI_Comm      comm        /* in  */);
```

进程root从sendbuf发送sendcount个sendtype的对象给comm中每个进程。comm中的每个进程接收recvcount个recvtype的对象到recvbuf中。大部分时间，sendtype和recvtype是一样的，sendcount和recvcount也是一样的。在任何情况下，明显进程root必须发送相同数目的对象给每个进程。

另外，MPI_Scatterv的语法如下：

```
int MPI_Scatterv(
      void*        sendbuf         /* in  */,
      int*         sendcounts      /* in  */,
      int*         displacements   /* in  */,
      MPI_Datatype sendtype        /* in  */,
      void*        recvbuf         /* out */,
      int          recvcount       /* in  */,
      MPI_Datatype recvtype        /* in  */,
      int          root            /* in  */,
      MPI_Comm     comm            /* in  */);
```

320 MPI_Scatter中的参数sendcount被两个数组参数sendcounts和displacements所代替。这两个数组都含有comm_sz个元素：sendcounts[q]是发送给进程$q$的sendtype类型的对象数量。此外，displacements[q]表示发送给进程$q$的块的起始点。偏移量（displacement）的计算是以sendtype为单位的。所以，如果sendtype是MPI_INT，并且sendbuf有类型**int***，那么发送给进程$q$的数据的起始位置是：

```
sendbuf + displacements[q]
```

总的来说，displacements[q]表示进程$q$的数据在sendbuf中的偏移量。数据块的“单位”

宽度和 sendtype 的单位宽度相同。

类似地，MPI_Gatherv 可以用来代替 MPI_Gather：

```
int MPI_Gatherv(
      void*        sendbuf          /* in  */,
      int          sendcount        /* in  */,
      MPI_Datatype sendtype         /* in  */,
      void*        recvbuf          /* out */,
      int*         recvcounts       /* in  */,
      int*         displacements    /* in  */,
      MPI_Datatype recvtype         /* in  */,
      int          root             /* in  */,
      MPI_Comm     comm             /* in  */);
```

### 维持最佳回路

正如我们前面讨论并行化树搜索时所观测到的那样，让每个进程各自计算最佳回路很可能造成很多计算上的浪费，因为有些进程的最佳回路可能比其他进程的绝大多数回路代价大（见习题 6.21）。因此，当一个进程发现了一个新的最佳回路时，它应该把这个回路发送给其他进程。

首先要注意的是，当一个进程发现了一个新的最佳回路时，它只需要发送该最佳回路的代价给其他进程。每个进程在调用 Best_tour 时只使用当前最佳回路的代价。当然，当一个进程更新最佳回路时，它不关心上一个最佳回路所包含的城市；它只关心上一个最佳回路的代价要比新的最佳回路的代价大。

在进行树搜索的过程中，当一个进程需要发送新的最佳回路的代价给其他进程时，不能使用 MPI_Bcast，因为 MPI_Bcast 是阻塞式的，而且通信子里的每个进程都必须调用 MPI_Bcast。[321] 然而，在并行化树搜索中，知道需要执行广播操作的唯一进程是发现了新的最佳回路的进程。如果使用 MPI_Bcast，该进程就很有可能阻塞在调用处永不返回，因为它是唯一调用 MPI_Bcast 的进程。因此，我们需要设定新的发送方式，这种方式不会造成发送进程永久阻塞。

MPI 提供了多种可行的方案。最简单的方法就是让发现新的最佳代价的进程调用 MPI_Send，把数据发送给所有其他的进程：

```
for (dest = 0; dest < comm_sz; dest++)
   if (dest != my_rank)
      MPI_Send(&new_best_cost, 1, MPI_INT, dest, NEW_COST_TAG,
          comm);
```

这里，在程序中使用了一个特殊的标志，NEW_COST_TAG。它告诉接收进程，消息是一个新的代价，而不是其他类型的消息（比如，回路）。

目标进程可以周期性地检查新的最佳回路代价的到达。不能使用 MPI_Recv 测试消息，因为它是阻塞式的；如果一个进程调用

```
MPI_Recv(&received_cost, 1, MPI_INT, MPI_ANY_SOURCE, NEW_COST_TAG,
      comm, &status);
```

那么进程会阻塞直到一个匹配的消息到达。如果没有消息到达（例如没有进程发现新的最佳回路代价），那么进程就会挂起。幸运的是，MPI 提供了一个只测试是否有消息可用的函数；它不会真地接收消息。这个函数就是 MPI_Iprobe，它的语法如下：

```
int MPI_Iprobe(
            int         source        /* in  */,
            int         tag           /* in  */,
            MPI_Comm    comm          /* in  */,
            int*        msg_avail_p   /* out */,
            MPI_Status* status_p      /* out */);
```

它检查在通信子中来源为进程 source、标志为 tag 的消息是否可用。如果这条消息可用，将 * msg_avail_p 赋值为 true，将 * status_p 的成员赋予恰当的值。例如，将 status_p→MPI_SOURCE 赋予接收到消息来源的线程号。如果没有可用的消息，将 * msg_avail_p 赋值为 false。参数 source 和 tag 分别可以是通配符 MPI_ANY_SOURCE 和 MPI_ANY_TAG。所以，为了检查来自任一进程具有新代价的消息，可以调用：

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail, &status);
```

322 如果 msg_avail 是 true，那么可以通过调用 MPI_Recv 来接收新的代价。

```
MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
      NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
```

能够这样做的一个最合适的地方是在 Best_tour 函数里。在检查是否新的回路是最佳回路前，可以用程序 6-9 中的代码来测试来自其他进程的新回路代价。

**程序 6-9　检查最佳回路代价的 MPI 代码**

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
while (msg_avail) {
   MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
         NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
   if (received_cost < best_tour_cost)
      best_tour_cost = received_cost;
   MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
     &status);
}  /* while */
```

这段代码不断接收那些可用的新回路代价。每次接收的新代价如果比当前最佳回路代价小，那么变量 best_tour_cost 就会得到更新。

你是否发现这种策略存在问题呢？如果发送者没有缓冲功能，那么对 MPI_Send 的循环调用会导致发送进程阻塞直到提交一个相匹配的接收。如果其他所有进程已经完成了搜索，那么发送进程就会被挂起。所以循环调用 MPI_Send 是不安全的。

MPI 针对这个问题提供了多种解决方法：**缓冲发送**和**非阻塞发送**。下面讨论缓冲发送，习题 6.22 讨论非阻塞发送。

**发送模式和缓冲发送**

MPI 为发送操作提供了四种**模式**：**标准**（standard）、**同步**（synchronous）、**就绪**（ready）和**缓冲**（buffered）。不同的模式对于发送函数具有不同的语法。我们最开始了解的 MPI_Send 是标准模式。在不同模式下，MPI 实现决定是将消息的内容复制到自己的存储空间，还是一直阻塞到一个相匹配的接收操作被提交。在同步模式下，发送操作会一直阻塞到一个相匹配的接收操作被提交。在就绪模式下，在发送操作前会有一个相匹配的接收操作被提交，否则发送操作就是错误的。在缓冲模式下，如果一个相匹配的接收操作还没有被提交，那么 MPI 实现必须复制消息到本
323 地存储空间。本地存储空间必须由用户程序提供，而不是 MPI 实现。

每种模式有一个不同的函数：MPI_Send、MPI_Ssend、MPI_Rsend 和 MPI_Bsend，但这些函数的参数都与 MPI_Send 相类似。

```
int MPI_Xsend(
      void*          message       /* in */,
      int            message_size  /* in */,
      MPI_Datatype   message_type  /* in */,
      int            dest          /* in */,
      int            tag           /* in */,
      MPI_Comm       comm          /* in */);
```

函数 MPI_Bsend 使用的缓冲区必须通过调用函数 MPI_Buffer_attach 来指定：

```
int MPI_Buffer_attach(
      void*  buffer       /* in */,
      int    buffer_size  /* in */);
```

参数 buffer 是指向由用户程序分配的一块内存空间的指针，buffer_size 是这块空间以字节为单位的大小。之前“连接”的缓冲区可以由下面的函数回收：

```
int MPI_Buffer_detach(
      void*   buf_p       /* out */,
      int*    buf_size_p  /* out */);
```

参数 buf_p 返回的是之前分配的块内存的地址，*buf_size_p 是内存块的大小。调用 MPI_Buffer_detach 会阻塞直到 buffer 中的消息被传送完。因为 buf_p 是输出参数，所以应该采用操作符 &。例如：

```
char buffer[1000];
char* buf;
int buf_size;
. . .
MPI_Buffer_attach(buffer, 1000);
. . .
/* Calls to MPI_Bsend */
. . .
MPI_Buffer_detach(&buf, &buf_size);
```

在程序里的任意执行点，只能连接一个用户提供的缓冲区。所以，如果存在多个没有完成的缓冲发送操作，则需要预估缓冲数据的大小。当然，我们不能知道数据的确切大小，但我们知道在任何一次最佳回路的“广播”中，进行广播操作的进程调用 MPI_Bsend 的次数是 commm_sz -1，并且每次调用会发送一个 **int** 型整数。我们可以确定一次广播所需要的缓冲区的大小。被传送的数据需要的存储空间大小可以通过调用 MPI_Pack_size 来计算： 324

```
int MPI_Pack_size(
      int           count     /* in  */,
      MPI_Datatype  datatype  /* in  */,
      MPI_Comm      comm      /* in  */,
      int*          size_p    /* out */);
```

输出参数给出了一条消息所需空间大小的上界。然而这还不够。除了数据以外，一条消息还需要存储包括发送目的地、标志、通信子等信息，所以对于每条消息，都需要多余的开销。MPI 针对这些增加的开销给出了上界常数：MPI_BSEND_OVERHEAD。对于每次广播，下面的代码决定了所需存储空间的大小：

```
int data_size;
int message_size;
int bcast_buf_size;

MPI_Pack_size(1, MPI_INT, comm, &data_size);
message_size = data_size + MPI_BSEND_OVERHEAD;
bcast_buf_size = (comm_sz - 1)*message_size;
```

我们可以猜测广播次数的上界，然后乘以 bcast_buf_size，就能够得到需要连接的缓冲区的大小。

**输出最佳回路**

当程序结束时，我们希望打印出最佳回路和它的代价，所以需要把回路发送给进程 0。开始，我们似乎可以这么做：每个进程存储自己本地的最佳回路，在树搜索操作完成后，每个进

程把自己的最佳回路代价和全局最佳回路代价进行比较。如果两者相同，那么进程就把自己本地的回路发送给进程0。然而，这里存在几个问题。第一，TSP有向图很有可能存在多个相同代价的最佳回路，即不同的进程会发现不同的回路。如果这种情况发生，那么多个进程就会尝试发送它们自己的最佳回路给进程0，除了一个线程以外的其他所有线程都会因为调用MPI_Send而挂起。第二，可能有一个或多个进程没有接收到最新的最佳回路代价，它们会发送不是最佳回路给进程0。

我们可以用下面的方法来避免上面的问题：让每个进程存储自己的局部最佳回路，在所有进程完成搜索后都调用MPI_Allreduce，拥有全局最佳回路的进程也可以把它发送给进程0。下面的伪代码提供了详细的描述：

```
struct {
   int cost;
   int rank;
} loc_data, global_data;

loc_data.cost = Tour_cost(loc_best_tour);
loc_data.rank = my_rank;
MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC,
      comm);
if (global_data.rank == 0) return;
      /* 0 already has the best tour */
if (my_rank == 0)
   Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
   Send best tour to process 0;
```

325 这里的关键是调用MPI_Allreduce时使用的操作。如果只使用MPI_MIN，就能知道全局最佳回路的代价是多少，但不能知道是哪个线程拥有它。然而，MPI提供了一个预定义的操作符：MPI_MINLOC，它作用于一对参数值。第一个参数值是回路的代价，第二个参数值是最小代价的地址，即拥有最佳回路进程的进程号。如果超过一个进程拥有最小代价的回路，地址将是这些进程的进程号中的最小值。在调用MPI_Allreduce时，输入和输出缓冲区是有两个成员的结构。因为代价和进程号都是整数，所以成员都用**int**来表示。MPI提供了MPI_2INT来表示这种类型。调用MPI_Allreduce返回时，有下面两种选择：

- 如果进程0已经有了最佳回路，我们就简单地直接返回。
- 否则，拥有最佳回路的进程发送最佳回路给进程0。

**未接收的消息**

正如我们之前讨论的那样，某些消息很可能在并行化树搜索的过程中被漏掉而没有接收。一个进程可能在某些进程发现最佳回路前已经搜索完了自己的子树。这些不会导致程序输出错误的结果。必须等到所有的进程都调用了MPI_Allreduce，找到最佳回路的进程对MPI_Allreduce的调用才会返回。因此它会正确地返回最小代价的回路，而进程0会接收这个回路。

然而，未接收的消息会对调用MPI_Buffer_detach或MPI_Finalize造成一些影响。如果一个进程正在存储在缓冲区中但未被接收的消息，该进程可能会在这些调用上挂起。所以在尝试关闭MPI前，可以调用MPI_Iprobe尝试接收那些未被接收的消息。这些代码类似于在程序6-9中用于测试新的最佳回路代价的代码。参见程序6-9。实际上，唯一不使用集合通信发送的消息是发送给进程0的最佳回路，以及对最佳回路代价的广播。如果某些进程不参与操作，MPI集合通信就会将该进程挂起，所以只需要寻找未被接收的最佳回路。

在动态负载均衡的代码（后续会讨论到）中，还有其他的消息需要考虑，包括一些较大的消息。为了处理这种情况，可以使用由MPI_Iprobe返回的参数status来确定消息的大小，并在

必要的时候分配需要增加的存储资源（参见习题6.23）。 326

### 6.2.12 采用 MPI 和动态划分来实现树搜索

我们可以通过模拟 Pthreads 和 OpenMP 中动态划分的程序来实现 MPI 的动态树搜索。在 Pthreads 和 OpenMP 的程序中，线程的搜索函数会在每一轮 `while` 循环中都调用一个返回值为布尔型的函数 `Terminated`。当某个线程完成工作后，也就是它的栈为空时，该线程就会进入条件等待（Pthreads）或者忙等待（OpenMP），直到它接收到新的任务或者通知它已经没有要做的任务了。在第一种情况下，它会继续去搜索最佳回路，在第二种情况下，它会退出执行。栈中有超过两个记录的线程会分配一半的任务给处于等待状态的线程。

这些逻辑可以在分布式内存环境中得到模拟。当一个进程完成任务后，它进入忙等待，等待接收更多的工作或者接收到程序终止的信号。类似地，一个有任务可做的进程可以划分它的栈，把自己的工作分配给一个空闲进程。

关键的不同之处在于，这里没有一个可供进程等待任务的集中式存储点，所以进程在划分栈的时候不能简单地对一组正在等待的进程执行出队操作，或者简单地调用函数 `pthread_cond_signal`。它需要“知道”哪个进程处于等待任务的状态，才能发送任务给该进程。因此，不能简单地直接进入忙等待状态或者终止，完成任务的进程应该发送请求任务的消息给其他进程。如果这样做，当进程进入 `Terminated` 函数时，它会检查是否有来自其他进程的请求任务消息。如果有，并且该进程也没有可做的工作，就会返回一个拒绝的消息。因此，在分布式内存系统中，`Terminated` 函数的伪代码可以如程序6-10所示。

`Terminated` 函数在开始时会检查进程栈中所拥有的回路数（第1行）；如果它有至少两条“值得发送”的回路，它就调用 `Fulfill_request`（第2行）。`Fulfill_request` 检查进程是否已经接收到请求任务的消息。如果没有接收到请求，就直接返回。在这种情况下，当从 `Fulfill_request` 返回后，它从 `Terminated` 返回并继续搜索。

如果调用进程没有至少两条值得发送的回路，`Terminated` 就调用 `Send_rejects`（第5行），检查是否有请求任务的消息，并发送一个“没有工作”（no work）的回复给每个请求任务的进程。此后，`Terminated` 检查调用进程是否有工作可做，如果有（栈非空），它就会返回并继续搜索。

当调用进程没有工作可做的时候，事情会变得很有趣（第9行）。如果在通信子里只有一个进程（`comm_sz = 1`），那么进程从 `Terminated` 返回并退出执行。如果有超过一个进程存在，那么进程宣布自己没有任务可做了（第11行）。这是“分布式终止检测算法”的部分实现，我们 327
会在后面继续讨论。目前，使用共享内存的终止检测算法是不可用的，因为它不能保证存储无任务可做的进程数目的变量是最新值。

**程序6-10 使用 MPI 实现动态划分的 TSP 问题中的 `Terminated` 函数**

```
1  if (My_avail_tour_count(my_stack) >= 2) {
2     Fulfill_request(my_stack);
3     return false;  /* Still more work */
4  } else { /* At most 1 available tour */
5     Send_rejects();  /* Tell everyone who's requested */
6                      /* work that I have none         */
7     if (!Empty_stack(my_stack)) {
8        return false;  /* Still more work */
9     } else {  /* Empty stack */
10       if (comm_sz == 1) return true;
11       Out_of_work();
12       work_request_sent = false;
```

```
13        while (1) {
14           Clear_msgs();  /* Msgs unrelated to work, termination */
15           if (No_work_left()) {
16              return true;  /* No work left.  Quit */
17           } else if (!work_request_sent) {
18              Send_work_request();  /* Request work from someone */
19              work_request_sent = true;
20           } else {
21              Check_for_work(&work_request_sent, &work_avail);
22              if (work_avail) {
23                 Receive_work(my_stack);
24                 return false;
25              }
26           }
27        }  /* while */
28     } /* Empty stack */
29  } /* At most 1 available tour */
```

在进入无限循环 **while**（第 13 行）前，设置变量 work_request_sent 为 false（第 12 行）。正如名字所示，这个变量表达是否给其他进程发送请求任务的消息；如果有，在发送下一条请求给其他进程前，持续等待来自它之前发送消息的目的进程所发来的任务，或者等待一条该进程响应的“没有可分配任务”的消息。

**While**(1)循环是 OpenMP 中忙等待循环的分布式内存版本。进程持续等待直到接收到分配的任务或者搜索工作已经完成的消息。

在进入 **while**(1)循环后，可以在第 14 行处理任何未被接收的消息。可能会接收到对最佳回路代价的更新，也可能接收到请求任务的消息。对于那些请求任务的进程，返回给其“没有可分
328 配任务”的消息是很有必要的，这样才能保证在没有任务的情况下它们不会一直处于等待状态。这对于最佳回路代价的更新也是个不错的主意，因为这样做可以释放发送进程消息缓冲区的空间。

在处理完未接收的消息后，迭代操作会按照以下步骤继续进行：

- 搜索已经完成，退出执行第 15 ~ 16 行。
- 没有发送任何任务请求，可以选择一个进程并发送请求任务的消息（第 17 ~ 19 行）。下面将进一步讨论发送请求给哪个进程。
- 有一个未完成任务请求（第 21 ~ 25 行）。所以我们测试该请求是否完成或者拒绝。如果已经完成，那么就接收新的工作并返回继续搜索；如果接收到拒绝的消息，就设置 work_request_sent 为 false 并继续执行循环操作。如果请求既没有完成也没有拒绝，那么就继续在 **while**(1)循环里执行。

下面给出一些函数的详细信息：

My_avail_tour_count

函数 My_avail_tour_count 返回进程栈的大小。它还可以使用“截止长度”。当一个部分回路已经访问了绝大多数城市时，这棵子树没完成的任务就很少了。因为发送一个部分回路操作的代价是很大的，可以尝试只发送边数少于截止部分的回路。在习题 6.24 中，我们考察截止对程序的全局运行时间的影响。

Fulfill_request

如果一个进程有足够的任务可做，即可以有效地分离它的栈，那么就调用函数 Fulfill_request（第 2 行）。Fulfill_request 使用 MPI_Iprobe 测试来自其他进程的任务请求。如果存在这样的请求，就接收它，并分离自己的栈，分配自己的工作给发送请求任务消息的进程；如果没有这样的请求，进程就继续往下执行。

### 分离栈

函数 Split_stack 是由 Fulfill_request 调用的。它所用的算法与 Pthreads 和 OpenMP 是一样的，即收集少于 split_cutoff 个城市的部分回路，并发送给请求任务的进程。然而，在共享内存程序中，简单地从旧栈复制回路（指针）到新栈。不幸的是，这涉及新栈的指针，这样的数据结构不能简单地发送给其他进程（见习题 6.25）。因此，MPI 版本的 Split_stack 把新栈的内容打包到连续的存储空间，然后把地址连续的内存块发送出去，并由接收者解包到自己的新栈里。

MPI 提供了函数 MPI_Pack，用于打包数据到地址连续的内存缓冲区。它还提供了函数 MPI_Unpack，用于解包数据。在习题 6.20 中，我们简单地介绍了它们。回顾一下它们的句法： 329

```
int MPI_Pack(
      void*          data_to_be_packed    /* in     */,
      int            to_be_packed_count   /* in     */,
      MPI_Datatype   datatype             /* in     */,
      void*          contig_buf           /* out    */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      MPI_Comm       comm                 /* in     */);

int MPI_Unpack(
      void*          contig_buf           /* in     */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      void*          unpacked_data        /* out    */,
      int            unpack_count         /* in     */,
      MPI_Datatype   datatype             /* in     */,
      MPI_Comm       comm                 /* in     */);
```

MPI_Pack 把 data_to_be_packed 中的数据打包到 contig_buf 里。参数 * position_p 表示在 contig_buf 中所处的位置。当该函数被调用时，它应该指向 data_to_be_packed 被加入前 contig_buf 可用的第一个位置。该函数返回后，应该指向 data_to_be_packed 被加入后 contig_buf 第一个可用的位置。

MPI_Upack 的操作与 MPI_Pack 正好相反。它把 contig_buf 里的数据解包到 unpacked_data。调用该函数时，* position_p 应该是未解包前 config_buf 的第一个可用位置。函数返回时，* position_p 应该是数据解包后 contig_buf 的下一个可用的位置。

例如，假设一段程序包含下面的定义：

```
typedef struct {
   int* cities; /* Cities in partial tour          */
   int count;   /* Number of cities in partial tour */
   int cost;    /* Cost of partial tour             */
} tour_struct;
typedef tour_struct* tour_t;
```

然后，我们用下面的代码发送类型为 tour_t 的变量：

```
void Send_tour(tour_t tour, int dest) {
   int position = 0;

   MPI_Pack(tour->cities, n+1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Pack(&tour->count, 1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Pack(&tour->cost, 1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Send(contig_buf, position, MPI_PACKED, dest, 0, comm);
}  /* Send_tour */
```

类似地，我们可以用下面的代码接收类型为 tour_t 的变量： 330

```
void Receive_tour(tour_t tour, int src) {
   int position = 0;

   MPI_Recv(contig_buf, LARGE, MPI_PACKED, src, 0, comm,
         MPI_STATUS_IGNORE);
   MPI_Unpack(contig_buf, LARGE, &position, tour->cities, n+1,
         MPI_INT, comm);
   MPI_Unpack(contig_buf, LARGE, &position, &tour->count, 1,
         MPI_INT, comm);
   MPI_Unpack(contig_buf, LARGE, &position, &tour->cost, 1,
         MPI_INT, comm);
}  /* Receive_tour */
```

在 MPI 程序中，用于发送和接收打包缓冲区的数据类型是 MPI_PACKED。

Send_rejects

函数 Send_rejects（第 5 行）类似于寻找新的最佳回路的函数。它使用 MPI_Iprobe 搜索请求任务的消息。这样的消息可以用一个特别的标志来识别，例如，WORK_REQ_TAG。找到这样的消息后就接收该消息，然后把“没有适合分配的工作”的信息回复给请求任务的进程。此时，标志能告知接收者消息的含义即可，请求任务的消息和表明无合适工作的消息都可以仅包含 0 个元素。即使这样的消息没有什么实际的内容，信封也需要占据空间，这些消息也需要被接收。

**分布式终止检测**

函数 Out_of_work 和 No_work_left（第 11 行和第 15 行）实现了终止检测算法。正如前面提及的，在共享内存程序中使用的终止检测算法在这里会出现问题。为了说明这点，假设每个进程存储了一个变量 oow，用于表示没有工作可做的进程数。在程序开始时，oow 被赋值为 0。每次某个进程完成了任务，就会发送一条消息告知其他所有进程它的状态，使得所有的进程都会对它自己的 oow 副本进行加 1 操作。类似地，当一个进程从另一个进程那里接收到工作时，它就需要给每个进程发送消息来告知此事，然后每个进程都会对各自的 oow 副本执行减 1 操作。如果假设有 3 个进程，进程 2 仍然有任务要做，但是进程 0 和进程 1 已经完成了各自的任务。考虑如表 6-10 所示的事件发生顺序。

**表 6-10　导致错误的终止事件**

| 时间 | 进程 0 | 进程 1 | 进程 2 |
|---|---|---|---|
| 0 | 无工作，通知进程 1 和进程 2，oow = 1 | 无工作，通知进程 0 和进程 2，oow = 1 | 工作，oow = 0 |
| 1 | 发送请求给进程 1，oow = 1 | 发送请求给进程 2，oow = 1 | 接收来自进程 1 的通知，oow = 1 |
| 2 | oow = 1 | 接收来自进程 0 的通知，oow = 2 | 接收来自进程 1 的请求，oow = 1 |
| 3 | oow = 1 | oow = 2 | 发送工作给进程 1，oow = 0 |
| 4 | oow = 1 | 接收来自进程 2 的工作，oow = 1 | 接收来自进程 0 的通知，oow = 1 |
| 5 | oow = 1 | 通知进程 0，oow = 1 | 工作，oow = 1 |
| 6 | oow = 1 | 接收来自进程 0 的请求，oow = 1 | 无工作，通知进程 0 和进程 1，oow = 2 |
| 7 | 接收来自进程 2 的通知，oow = 2 | 发送工作给进程 0，oow = 0 | 发送请求给进程 1，oow = 2 |
| 8 | 接收来自进程 1 的通知，oow = 3 | 接收来自进程 2 的通知，oow = 1 | oow = 2 |
| 9 | 退出 | 接收来自进程 2 的请求，oow = 1 | oow = 2 |

这里发生的错误是：进程 1 发送给进程 0 的工作丢失了。原因是进程 0 在收到进程 1 已接收到工作的通知前，先收到进程 2 没有任务的通知。这种情况看似不会发生，但是却存在发生
331 的可能。例如，进程 1 被操作系统中断，以至于直到进程 2 发送的消息传送后，进程 1 的才能传送。

尽管 MPI 保证了两条从进程 A 发送到进程 B 的消息会按照它们发送的顺序被接收，但它却不能保证不同进程发送消息的接收顺序。考虑到不同的进程会由于各种原因有不同的执行速度，出现问题是很容易理解的。

分布式终止检测是一个具有挑战性的问题，目前，在开发保证正确检测终止状态的算法上已经有人做了很多工作。从概念上讲，这些算法中最简单的一种是追踪一个守恒量，这个守恒量可以精确测量。我们称这个守恒量为能量（energy），因为能量是守恒的。在程序的起始处，每个进程有 1 个单位的能量。当一个进程完成了任务，就把自己的能量发送给进程 0。当一个进程完成了一次请求任务的操作，就把自己的能量一分为二，自己留一份，另外一份发送给接收任务的进程。因为能量是守恒的，起始的份额是 comm_sz 个单位，所以程序会在进程 0 接收到 comm_sz 个单位的能量后终止执行。 332

函数 Out_of_work 在被除了进程 0 以外的进程执行时会发送能量给进程 0。进程 0 可以用变量 received_energy 记录能量数。函数 No_work_left 也依赖于是由进程 0 调用还是由其他进程调用。如果进程 0 调用，就接收由 Out_of_work 发送的消息，并调整变量 received_energy。如果 received_energy 等于 comm_sz，进程 0 就发送一个终止消息（具有特殊标志）给每个进程。另一方面，非 0 进程会测试是否有一个标志为终止的消息。

这里比较重要的一点是，要确保没有能量流失；如果采用 **float** 或者 **double** 类型来表示能量，那么一定会出现问题，因为在除法操作时会引起下溢。因为能量的数量可以用普通分数来表达，所以可以精确地为每个进程用一对定点数来表达能量数。分母总是 2 的幂，所以采用以 2 为底的算法。但对一个大问题来说，分子仍然可能会溢出。然而，如果这真成了问题，还可以使用提供了任意精度有理数的类库（例如 GMP [21]）。另外一种解决方法在习题 6.26 中进行讨论。

**发送请求任务的消息**

一旦决定了给哪个进程发送请求，就可以发送一个长度为 0，标志为“请求工作”的消息。然而，在选定目标进程上有以下可能。

1）用时间片轮转的方式依次通过每个进程。以(my_rank +1)% comm_sz 为起点，每来一个新的请求就把目标进程号加 1。这样做可能存在的问题是，两个进程会进入“同步”，并重复给相同的目的地发送请求任务的消息。

2）在进程 0 维护一个全局目标进程的信息。当某个进程完成任务后，它首先从进程 0 请求当前全局目标进程的值。进程 0 在每个请求后对该值执行加 1 操作。这样就避免了多个进程给同一目的地发送消息，但很明显进程 0 会成为瓶颈。

3）每个进程使用一个随机数生成器来产生目的地。当然还是有可能发生多个进程同时向一个进程发送请求，不过随机生成目标进程号能降低多个进程重复请求同一个进程的可能。

我们在习题 6.29 里分析这几种方案，也可以参考 [22] 对这些选项的分析。

**检查和接收任务**

一旦有进程发送了请求任务的消息，该进程就需要重复地检查是否有目标进程的回复。实际上，微妙之处在于，发送进程需要检查接收进程返回消息的标志到底是“有任务可以分配”还是“无任务可以分配”。如果发送进程只是简单地检查是否有来自接收进程的消息，那就有可能被 333
“误导”，导致永远都接收不到发送过来的任务。例如，一个来自目标进程请求任务的消息，有可能会掩盖有任务消息的存在。

因此函数 Check_for_work 首先要探测是否有表示“有任务可分配”的消息，如果没有这样的消息，就需要探测是否有表示“没有任务可分配”的消息。如果有任务可分配，函数 Receive_work 可以接收含有任务的消息，并解包相应的内容到进程的栈中。同时要注意，还需要解包来自

目标进程的能量值。

**MPI 程序的性能**

表 6-11 显示了数量为 15 个城市的两个 TSP 问题的 MPI 程序的性能，这两个问题与我们之前用 Pthreads 和 OpenMP 测试过的问题相同。运行时间以秒为单位，括号中的数字代表动态实现时分离栈的次数。取得这些结果的系统不同于之前 Pthreads 方式下所用的系统。当然我们也把这个系统下用 Ptheads 实现该问题所获得的性能展示出来，所以两组数据可以用来对比。这个系统的结点只有 4 个核，所以 Pthreads 的结果不包括 8 核和 16 核的情况。MPI 实现时，参数 cutoff 设置为 12。

表 6-11 树搜索的 MPI 和 Pthreads 实现的性能（单位：秒）

| 线程/进程 | 第一个问题 | | | | | | 第二个问题 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 静态 | | 动态 | | | | 静态 | | 动态 | | | |
| | Pth | MPI | Pth | | MPI | | Pth | MPI | Pth | | MPI | |
| 1 | 35.8 | 40.9 | 41.9 | (0) | 56.5 | (0) | 27.4 | 31.5 | 32.3 | (0) | 43.8 | (0) |
| 2 | 29.9 | 34.9 | 34.3 | (9) | 55.6 | (5) | 27.4 | 31.5 | 22.0 | (8) | 37.4 | (9) |
| 4 | 27.2 | 31.7 | 30.2 | (55) | 52.6 | (85) | 27.4 | 31.5 | 10.7 | (44) | 21.8 | (76) |
| 8 | | 35.7 | | | 45.5 | (165) | | 35.7 | | | 16.5 | (161) |
| 16 | | 20.1 | | | 10.5 | (441) | | 17.8 | | | 0.1 | (173) |

这个系统的每一个结点是一个小型的共享内存系统，因此采用共享变量形式的通信比分布式内存通信快。所以在各种情况下，Pthreads 实现的性能好于 MPI 实现，是可以理解的。

在 MPI 实现里，分离栈操作的代价比较高；除了通信开销以外，打包和解包也很耗时。所以
334 对于采用较少进程的小问题，静态 MPI 并行化实现要优于动态并行化。然而，8 进程和 16 进程的结果表明，如果问题大到足够能保证使用多个进程，那么动态 MPI 程序的可扩展性更高，性能也更出色。例子中对于城市数为 17、进程数为 16 的 TSP 问题印证了这一点：动态 MPI 实现的运行时间是 296 秒，静态实现的运行时间为 601 秒。

对于第二个问题，在 16 个进程的情况下，0.1 秒的执行时间也不能真正代表超线性的加速比。更准确地说，是初始化分配任务让某个进程发现最佳回路的速度比进程较少时要快了许多，动态分割帮助进程之间维持更均衡的负载。

## 6.3 忠告

在开发针对 $n$ 体问题和 TSP 的解决方案时，选择串行算法是因为它更容易理解，对它的并行化更直接。但无论如何我们不会因为串行算法是最快的，或者它可以解决最大的问题而选择串行算法。因此，我们不应该假设串行或并行解决方案是最好的。目前最好的算法，$n$ 体问题可以参见［12］，并行化树搜索可以参见［22］。

## 6.4 选择哪个 API

如何判定 MPI、Pthreads、OpenMP 中哪一个是针对某个应用程序最佳的 API 呢？一般来说，需要考虑许多因素，结果也不是特别明显。然而，下面一些要点是我们需要注意的：

第一步，决定采用分布式内存还是共享内存。为了做出决定，先考虑应用程序所需内存的数量。一般情况下，分布式内存系统比共享内存系统提供更多的内存空间，所以如果所需内存较大，可以考虑使用 MPI 来编写应用程序。

即使共享内存系统能满足问题的内存需求，依然可以考虑采用 MPI。因为分布式内存系统有

比共享内存系统多得多的高速缓存，可以想象，在共享内存系统中需要执行大量主存存取操作的问题，会转化成在分布式内存系统中执行大量存取高速缓存的操作，结果会得到更优的全局性能。

然而，即使通过利用分布式内存系统中较大的总缓存量得到很大的性能提升，如果已经有了一个大而复杂的串行程序，开发它对应的共享内存程序也是有意义的。通常情况下，在共享内存 335
系统中可以比在分布式内存系统中重用更多的串行代码。串行数据结构可以更容易地部署在共享内存系统中。在这种情况下，开发基于共享内存系统的程序会轻松些。这对于 OpenMP 程序来说，确实如此，因为很多串行代码可以通过 OpenMP 指令很轻松地并行化。

另一个需要考虑的是并行化程序的通信需求。如果进程/线程之间通信较少，MPI 程序会很容易开发，并且具备良好的扩展性。对于另一个极端情况，进程/线程需要很好地协调时，分布式内存程序会在扩展到一定程度时出现问题，此时选择共享内存程序会好一些。

如果你确定共享内存更好，那就需要考虑并行化程序的细节。正如我们之前所关注的，如果是一个大而复杂的串行程序，就需要先考虑是否适合采用 OpenMP。例如，如果部分可以用 `parallel` 指令来并行化，OpenMP 会比 Pthreads 更易于使用。另一方面，如果程序涉及线程之间复杂的同步（例如读写锁、线程等待被其他线程唤醒），那么 Pthreads 会更容易使用些。

## 6.5　小结

在本章中，我们观察了针对两个不同问题的串行和并行解决方案：$n$ 体问题和 TSP 树搜索方案。对每一个例子，我们一开始都是研究问题并提出该问题的串行解决方案。然后采用 Foster 方法设计一个并行解决方案。根据 Foster 方法，我们实现了分别采用 Pthreads、OpenMP 和 MPI 的解决方案。在开发针对 $n$ 体问题的简化版本的 MPI 解决方案时，我们发现“显而易见”的解法在实现上很难，并且需要大量的通信。因此我们转而采用“环形传递”算法，该算法被证明更容易实现，可扩展性更好。

对于并行化树搜索的动态分割解法，三种 API 分别采用了不同的方法。在 Pthreads 实现里，使用一个条件变量来控制线程之间新任务的通信，以及线程的终止。OpenMP 没有提供类似于 Pthreads 中条件变量的对象，所以使用了忙等待机制。在 MPI 的实现里，因为所有的数据都是局部的，所以需要使用更复杂的机制重新分配任务。为了正确地重新分配任务，一个无任务可做的进程会进入忙等待循环，在该循环中它会发送请求任务的消息，并等待响应其请求任务的消息，或者进程终止的消息。 336

我们看到，在分布式内存环境里，进程之间互相发送任务，要决定何时停止执行不是一个简单的问题。我们采用了一个比较直接的解决方案处理分布式终止检测问题，即在程序执行的过程中使用固定数量的“能量”。在进程无任务可做时，它们会发送自己的能量给进程 0，当进程发送任务给其他进程的同时，它们会发送当前一半的能量给接收者。因此，当进程 0 发现它拥有全部能量时，就意味着没有任务可做了，也就可以发送终止消息了。

接下来，我们简单分析了如何选择 API。第一个要考虑的问题是使用共享内存还是分布式内存。为了做出决定，需要考察应用程序所需内存的大小，以及进程/线程之间的通信量。如果所需内存很大，或者分布式内存版本可以利用高速缓存，那么分布式内存程序可能会运行得更快。相反地，如果有较多的通信，共享内存程序会运行得更快些。

对于如何选择 OpenMP 和 Pthreads，如果已经有串行程序，并且可以通过 OpenMP 的指令并行化，那么 OpenMP 会是一个较好的选择。然而，如果需要线程间有复杂的同步（例如，读写锁、线程信号量），那么 Pthreads 会更易于使用。在开发这些程序的过程中，我们也学到了更多关于 Pthreads、OpenMP 和 MPI 的知识。

### 6.5.1 Pthreads 和 OpenMP

在树搜索问题中，我们需要在更新最佳回路之前，检查当前的最佳回路。在并行化树搜索的 Pthreads 和 OpenMp 实现里，更新最佳回路会导致竞争冲突。“检查锁的条件”和“更新锁的条件”的组合会导致一个问题：上锁的条件（例如，最佳回路的代价）可能会在第一次检查锁条件的时间和获得锁的时间之间改变。因此，线程在取得锁之后依然要再检查上锁的条件，所以更新最佳回路的伪代码应该类似下面：

```
if (new_tour_cost < best_tour_cost) {
   Acquire lock protecting best tour;
   if (new_tour_cost < best_tour_cost)
      Update best tour;
   Relinquish lock;
}
```

记住在 Pthreads 里有一个非阻塞版本的 pthreads_mutex_lock，称为 pthread_mutex_trylock。这个函数测试互斥量是否可用。如果可用，就取到锁，返回 0 值；如果不可用，它不
337 会一直等待其变成可用状态，它会立即返回非 0 值。

OpenMP 中，类似于 pthread_mutex_trylock 的是 omp_test_lock。然而，它的返回值与 pthread_mutex_trylock 正好相反。

当一个单独的线程需要执行一个结构化代码时，OpenMP 提供了一些解决方法：

```
if (my_rank == special_rank) {
   Execute action;
}
```

采用 single 指令：

```
#     pragma omp single
      Execute action;

      Next action;
```

系统会选择一个单独的线程去执行该动作。其他线程会在进入 Next action 前，一直等待在隐含的路障处。采用 master 指令：

```
#     pragma omp master
      Execute action;

      Next action;
```

主线程（线程 0）会执行该动作。然而，与 single 指令不同，在块 Execute action 后没有隐含的路障，组内其他线程会立即执行 Next action。当然，如果在继续执行前需要一个路障，可以在完成结构化块 Execute action 后增加一个路障。在习题 6.6 里，OpenMP 提供了一个 nowait 的子句，可以用于修改 single 指令：

```
#     pragma omp single nowait
      Execute action;

      Next action;
```

当这个子句被加入后，系统选中的用于执行该动作的线程与以前一样，但是组内其他线程却不会等待，它们会立即执行 Next action。nowait 子句还可以用于修改 parallel **for** 和 **for** 指令。

### 6.5.2 MPI

我们已经学习了 MPI 的一部分知识。某些集合通信函数使用输入缓冲区和输出缓冲区，可以使用参数 MPI_IN_PLACE 来保证输入和输出缓冲区是同一个内存区域。这样的实现可以节省内
338 存，同时避免从输入缓冲区到输出缓冲区的复制操作。

函数 MPI_Scatter 和 MPI_Gather 可分别用于分发一组数据到所有进程和收集分散的数据到一个单独的数组。然而，它们只能在每个进程分发或收集的数据量一致的情况下使用。如果需要分配不同数量的数据给每个进程，或者从每个进程收集数量不同的数据，那么可以分别使用 MPI_Scatterv 和 MPI_Gatherv：

```
int MPI_Scatterv(
        void*        sendbuf          /* in  */,
        int*         sendcounts       /* in  */,
        int*         displacements    /* in  */,
        MPI_Datatype sendtype         /* in  */,
        void*        recvbuf          /* out */,
        int          recvcount        /* in  */,
        MPI_Datatype recvtype         /* in  */,
        int          root             /* in  */,
        MPI_Comm     comm             /* in  */);

int MPI_Gatherv(
        void*        sendbuf          /* in  */,
        int          sendcount        /* in  */,
        MPI_Datatype sendtype         /* in  */,
        void*        recvbuf          /* out */,
        int*         recvcounts       /* in  */,
        int*         displacements    /* in  */,
        MPI_Datatype recvtype         /* in  */,
        int          root             /* in  */,
        MPI_Comm     comm             /* in  */);
```

MPI_Scatterv 里的参数 sendcounts 和 MPI_Gatherv 里的参数 recvcounts 是有 comm_sz 个元素的数组。它们表示的是要传输数据的大小（以 sendtype/recvtype 为单位）。参数 displacements 也是有 comm_sz 个元素的数组。它们表达的是要传输数据的偏移量（以 sendtype/recvtype 为单位）。

这里有一个特别的操作符，MPI_MIN_LOC，可用在 MPI_Reduce 和 MPI_Allreduce 中。它作用于多对数值，并返回一对数值。如果数据对是：

$$(a_0,b_0),(a_1,b_1),\cdots,(a_{comm\_sz-1},b_{comm\_sz-1}),$$

假设 $a$ 是 $a_i$ 中的最小值，$q$ 是 $a$ 所出现在的进程中最小的进程号。那么 MPI_MIN_LOC 操作符返回 $(a_q, b_q)$。使用它，不仅可以用来求解最小代价回路，通过确定 $b_i$ 的进程号，还能确定拥有最小代价回路的进程。

在并行化树搜索的两个 MPI 实现里，我们重复使用 MPI_Iprobe：

```
int MPI_Iprobe(
          int         source        /* in  */,
          int         tag           /* in  */,
          MPI_Comm    comm          /* in  */,
          int*        msg_avail_p   /* out */,
          MPI_Status  status_p      /* out */);
```

使用它检查是否有一条来自 source 的标志为 tag 的可接收消息。如果有，将 msg_avail_p 赋 339
值为 true。注意 MPI_Iprobe 不会真的接收消息，但是如果有消息可用，可以调用 MPI_Recv 接收消息。参数 source 和 tag 分别可以是通配符 MPI_ANY_SOURCE 和 MPI_ANY_TAG。例如，我们经常想要检查是否有进程发送了带有新的最佳回路的消息，就可以进行以下调用：

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
```

如果有消息可用，它的来源由参数 * status_p 返回。因此，status.MPI_SOURCE 可以用于接收消息：

```
MPI_Recv(&new_cost, 1, MPI_INT, status.MPI_SOURCE, NEW_COST_TAG,
      comm, MPI_STATUS_IGNORE);
```

有多种情况希望发送函数直接返回，而不考虑消息是否已经真地发送出去了。MPI 里的一种做法是使用**缓冲发送模式**。在缓冲发送时，用户程序通过调用 MPI_Buffer_attach 分配存储空间。然后当程序调用 MPI_Bsend 发送消息时，消息既可能立即发送，也可能被复制到用户程序提供的缓冲区里。这两种情况下调用都会返回而不会阻塞。在程序不需要使用缓冲发送模式时，可以调用 MPI_Buffer_detach 来恢复缓冲区。

MPI 也提供了其他三种发送模式：**同步**（synchronous）、**标准**（standard）、**就绪**（ready）。同步发送不会缓冲数据；同步发送函数 MPI_Ssend 在接收者开始接收数据前不会返回。在就绪发送（MPI_Rsend）下，在匹配的接收者没有准备好之前，发送操作是错误的。一般的 MPI_Send 函数称为标准发送模式。

在习题 6.22 里，我们设计了一个缓冲发送模式的替代方法：非阻塞发送。正如名字所表示的那样，非阻塞发送模式不考虑消息是否已经发送出去，它直接返回。然而，要完成发送操作，需要调用一个函数，这个函数是多个等待非阻塞操作完成的函数中的一个。我们还有非阻塞接收函数。

因为一个系统的地址空间一般与另外一个系统是不相关的，用 MPI 消息发送指针是不可行的。如果数据结构中有嵌入指针成员，那么 MPI 就会提供函数 MPI_Pack 将指针所指向的数据存储到一块连续的缓冲区内，然后把这块缓冲区发送出去。类似地，函数 MPI_Upack 的功能正好
340 相反。它们的句法是：

```
int MPI_Pack(
      void*          data_to_be_packed    /* in     */,
      int            to_be_packed_count   /* in     */,
      MPI_Datatype   datatype             /* in     */,
      void*          contig_buf           /* out    */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      MPI_Comm       comm                 /* in     */);

int MPI_Unpack(
      void*          contig_buf           /* in     */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      void*          unpacked_data        /* out    */,
      int            unpack_count         /* in     */,
      MPI_Datatype   datatype             /* in     */,
      MPI_Comm       comm                 /* in     */);
```

使用它们的关键是参数 position_p。调用 MPI_Pack 时，它指向 contig_buf 中第一个可用位置。所以，开始打包数据时，*position_p 应设置为 0。当 MPI_Pack 返回时，*position_p 是打包数据后的第一个位置。因此，可以连续调用 MPI_Pack，把一个数据结构的连续成员打包到单独的缓冲区内。当打包的缓冲区被接收时，数据可以用同样的方式解包。需要注意的是，当调用 MPI_Pack 打包的缓冲区被发送后，发送和接收的数据类型就应该是 MPI_PACKED。

## 6.6 习题

6.1 在串行 $n$ 体问题算法的每一次迭代中，首先计算每一个粒子所受的总作用力，然后为每一个粒子计算它们的位置和速度。是否可以重新组织代码，使得在每一次迭代中，在进入下一个粒子的计算前，能够先完成对一个粒子的所有计算。或者说，是否可以采用以下伪代码：

```
for each timestep
   for each particle {
      Compute total force on particle;
      Find position and velocity of particle;
      Print position and velocity of particle;
   }
```

如果可以，还需要做其他哪些修改？如果不可以，为什么不可以？

6.2 运行基本的串行 $n$ 体问题算法 1000 个时间步长，一个时间步长为 0.05，无输出，也没有内部产生的初始变量。粒子的数目为 500 ~ 2000 个。随着粒子数目的增加，运行时间是如何改变的？如果程序运行 24 小时，你能推断和预测这个算法能够处理多少个粒子吗？ 341

6.3 用 OpenMP 或者 Pthreads 来并行化 $n$ 体问题的简化版本算法，使用一条 `critical` 指令（OpenMP）或者一个互斥量（Pthreads）来保护 `force` 数组。通过并行化内层 **`for`** 循环来并行化算法的剩下部分。与串行算法相比，这个代码的性能如何？解释你的回答。

6.4 用 OpenMP 或者 Pthreads 来并行化 $n$ 体问题的简化版本算法，并且对每一个粒子使用一个锁/互斥量。锁/互斥量是用来对 `force` 数组的更新实施保护的。通过并行化内层 **`for`** 循环来并行化解法的剩下部分。与串行算法相比，这个代码的性能如何？解释你的回答。

6.5 在这个共享内存 $n$ 体问题的简化版本算法中，如果在两个计算作用力阶段都使用块划分方法，那么第二个阶段的循环语句可以改变，将 **`for`** `thread` 循环的终止变量从 `thread_count` 改变为 `my_rank`。即代码：

```
#       pragma omp for
        for (part = 0; part < n; part++) {
           forces[part][X] = forces[part][Y] = 0.0;
           for (thread = 0; thread < thread_count; thread++) {
              forces[part][X] += loc_forces[thread][part][X];
              forces[part][Y] += loc_forces[thread][part][Y];
           }
        }
```

改变为：

```
#       pragma omp for
        for (part = 0; part < n; part++) {
           forces[part][X] = forces[part][Y] = 0.0;
           for (thread = 0; thread < my_rank; thread++) {
              forces[part][X] += loc_forces[thread][part][X];
              forces[part][Y] += loc_forces[thread][part][Y];
           }
        }
```

解释为什么这个改变是可行的。运行改变后的程序，并将它的性能与原来实现了块划分的代码，以及在第一个计算作用力阶段使用循环划分的代码相比较。你能得出什么结论？

6.6 在讨论用 OpenMP 实现基本的 $n$ 体问题解法时，我们观察到输出语句隐含的路障同步是不需要的。因此可以在 `single` 指令增加一个 `nowait` 子句。另外，在两个 **`for`** `each particle q` 循环语句中为 **`for`** 指令增加 `nowait` 子句，也有可能消除循环语句隐含的路障同步。这样做会出现问题吗？解释你的回答。 342

6.7 对于 $n$ 体问题的简化版本算法的共享内存实现，我们看到不管是否降低了高速缓存的性能，对作用力的计算采用循环调度比采用块划分调度性能更好。用 OpenMP 和 Pthreads 的实现进行实验，观察不同的块 - 循环调度程序的性能。对你的系统，存在一个最优的块大小吗？

6.8 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 都是双精度的 $n$ 维向量，而 $\alpha$ 是一个双精度标量，我们称以下赋值：

$$\boldsymbol{y} \leftarrow \alpha\boldsymbol{x} + \boldsymbol{y}$$

为双精度的 $\alpha$ 乘以 $\boldsymbol{X}$ 再加上 $\boldsymbol{Y}$（Double precision Alpha times $\boldsymbol{X}$ plus $\boldsymbol{Y}$，DAXPY）编写一个 OpenMP 或者 Pthreads 程序，在主线程中生成两个随机 $n$ 维大数组、一个随机标量，它们的类型都是 **`double`**。然后由多个线程对随机生成的数据执行 DAXPY。用大数值的 $n$ 和不同数目的线程来运行程序，比较分别采

用块划分和循环划分对数组进行划分时，程序的性能。哪一种划分策略的性能较好？为什么？

6.9 编写一个 MPI 程序，程序的每一个进程都生成一个初始化好的、$m$ 维 **double** 型的大数组。然后对这些 $m$ 维数组执行 MPI_Allgather。比较下面两种情况下 MPI_Allgather 的性能，调用 MPI_Allgather 生成的全局大数组分别是：

a. 块分布

b. 循环分布

为了使用循环分布，可以从本书的网站上下载 cyclic_derived.c 的代码，并用这个代码生成的 MPI 数据类型作为调用 MPI_Allgather 生成的目标。例如，进行以下调用：

```
MPI_Allgather(sendbuf, m, MPI_DOUBLE, recvbuf, 1, cyclic_mpi_t,
      comm);
```

新生成的 MPI 数据类型叫 cyclic_mpi_t。

数组采用哪种分布，性能更好？为什么？注意不要把创建派生数据类型的开销计算进去。

6.10 考虑以下代码：

```
int n, thread_count, i, chunksize;
double x[n], y[n], a;
    . . .
#   pragma omp parallel num_threads(thread_count) \
       default(none) private(i) \
       shared(x, y, a, n, thread_count, chunksize)
    {
343 #     pragma omp for schedule(static, n/thread_count)
      for (i = 0; i < n; i++) {
         x[i] = f(i);  /* f is a function */
         y[i] = g(i);  /* g is a function */
      }
#     pragma omp for schedule(static, chunksize)
      for (i = 0; i < n; i++)
         y[i] += a*x[i];
   }  /*  omp parallel */
```

假定 n = 64，thread_count = 2，高速缓存行的大小是 8 个 **double**，每一个核有一个能存储 131 072 个 double 型数据的二级（L2）高速缓存。如果 chunksize = n/thread_count，那么在第二个循环中会出现多少次 L2 高速缓存缺失？你可以假定 x 和 y 都按照高速缓存行的边界对齐存储。即 x[0]和 y[0]都是它们所在的高速缓存行的第一个元素。

6.11 编写一个 MPI 程序，比较使用 MPI_IN_PLACE 和 MPI_Allgather 和每一个进程使用分离的发送和接收缓冲区的 MPI_Allgather 的性能。在运行单进程时，哪种 MPI_Allgather 运行得更快一些？在多进程的情况下又如何？

6.12 a. 修改 $n$ 体问题算法的基本 MPI 实现，对局部的位置信息采用分离的数组来存储。与原来的 $n$ 体问题算法相比，性能如何？（比较输入/输出关闭的性能。）

b. 修改 $n$ 体问题算法的基本 MPI 实现，对质量信息进行分散存储。对于程序中的通信部分需要做哪些修改？与原来的算法相比，性能如何？

6.13 使用图 6-6 为指导，画出简化的 $n$ 体算法 MPI 实现的进程间通信图。假设有 3 个进程，6 个粒子，粒子在进程之间的分配是循环分配。

6.14 修改简化的 $n$ 体算法的 MPI 版本。使用两次调用 MPI_Sendrecv_replace 来代替每一个阶段的环形传递。与只单独调用一次 MPI_Sendrecv_replace 相比较，这个实现性能如何？

6.15 MPI 程序一个常见的问题是将一个全局数组的下标转换为局部数组的下标，反之亦然。

a. 如果采用块划分，找一个从局部数组下标转换为全局数组下标的公式。

344 b. 如果采用块划分，找一个从全局数组下标转换为局部数组下标的公式。

c. 如果采用循环划分，找一个从局部数组下标转换为全局数组下标的公式。

d. 如果采用循环划分，找一个从全局数组下标转换为局部数组下标的公式。

6.16　在简化 $n$ 体算法实现中，使用了一个函数 First_index，给定分配给某个进程的一个粒子的全局下标，该函数能给出分配给另一个进程的下一个粒子的全局下标。函数的输入参数如下：

a. 分配给第一个进程的粒子的全局下标。

b. 第一个进程的进程号。

c. 第二个进程的进程号。

d. 进程的个数。

返回值是第二个粒子的全局下标。这个函数假定粒子在进程间循环分配。为 First_index 函数编写对应的 C 代码。（提示：考虑两种情况，第一个进程的进程号小于第二个进程的进程号，第一个进程的进程号大于第二个进程的进程号。）

6.17　a. 使用图 6-10，计算 4 个城市 TSP 问题在任意时刻栈中最大的记录数（提示：分支时尽可能地向左边深入下去）。

b. 画出解决 5 个城市 TSP 问题时生成的树的结构。

c. 计算搜索（b）所生成的树的栈中任意时刻最大的记录数。

d. 用上述问题的答案推导出 $n$ 个城市 TSP 问题栈中任意时刻最大的记录数。

6.18　广度优先搜索可以实现为一个迭代算法，其中要采用一个**队列**。队列是“先入先出”的链表数据结构，队列中的元素出队的顺序与入队的顺序是一致的。可以用队列来解决 TSP 问题，广度优先搜索的实现如下：

```
queue = Init_queue();  /* Create empty queue */
tour = Init_tour();    /* Create partial tour that visits
    hometown */
Enqueue(queue, tour);
while (!Empty(queue)) {
   tour = Dequeue(queue);
   if (City_count(tour) == n) {
      if (Best_tour(tour))
         Update_best_tour(tour);
   } else {                                          345
   for each neighboring city
      if (Feasible(tour, city)) {
         Add_city(tour, city);
         Enqueue(tour);
         Remove_last_city(tour);
      }
}
Free_tour(tour);
/* while !Empty */
```

这个算法虽然是正确的，但当城市数目超过 10 时，实现起来相当困难。为什么？

在 TSP 问题的共享内存解决方案中，可以使用广度优先搜索来创建初始回路列表，这些回路能分配给各个线程。

a. 修改上述代码，以便线程 0 可以用此代码来生成数目至少为 thread_count 的回路队列。

b. 一旦线程 0 生成了回路队列，编写出各个线程如何用队列里的一部分回路来初始化自己栈的伪代码。

6.19　修改静态树搜索的 Pthreads 实现，用读写锁来保护最佳回路的检查操作。调用 Best_tour 对最佳回路读锁，调用 Update_best_tour 对最佳回路写锁。用多个数据集来测试修改的代码，并分析这些改变会怎样影响总的运行时间。

6.20　假设进程/线程 A 的栈中有 $k$ 条回路。

a. 也许在 TSP 问题中实现栈分离最简单的一种策略是把 A 现有栈中的 $k/2$ 条回路出栈，并把它们压入新栈中。解释为什么这不是一个好策略。

b. 另一个简单的划分方法是根据栈中部分回路的代价来划分。最小代价的部分回路留在 A 中，下一个最小代价的部分回路到 new_stack 中，第三个最小代价的部分回路到 A 中，以次类推。这个

方法是一个好策略吗？请给出你的答案并详细分析。

c. b 方案的一个变体是使用边的平均代价。该变体中，A 的栈中回路排序的依据是，按照回路的代价除以回路中边的条数所得到的结果来排序。然后采用轮转方式来分配这些回路。排序第 1 的回路分配给 A，下一个排在第 2 位的回路分配给 new_stack，以次类推。这是一个好策略吗？请给出你的答案并详细分析。

在动态负载平衡的代码中实现上述三种方案，对这三种方案进行比较，并将它们与书中的方案进行比较，解释如何收集数据。

6.21 a. 修改 TSP 问题的静态 MPI 程序，使得每个进程都使用自己本地的最佳回路数据结构，直到它完成搜索为止。当所有的进程执行完成后，所有的进程执行一个全局归约操作来找出最小代价回路。该方案与静态实现相比较，性能上有什么不同？你能否找出一些例子，使得该方案的性能与原来的静态实现差别不大吗？

b. 创建一个 TSP 有向图，要求其分配给进程 1，2，…，comm_sz −1 的初始回路都有一条边，该边的代价要比进程 0 中任意一条完整回路的代价高。使用 comm_sz 个进程来解决这个问题时，有哪
346 些不同的实现方案？

6.22 MPI_Recv 和我们学习过的每一个发送操作都是阻塞式的。MPI_Recv 在没有收到消息前不会返回，不同的发送操作在消息没有发送或没有缓冲前不会返回的。因此，当这些操作返回时，就能得知消息缓冲区参数的状态。对于 MPI_Recv，消息缓冲区含有接收到的消息，至少表示目前没有发生错误。而对于发送操作，消息缓冲区可以被重用。MPI 还提供了上述函数的**非阻塞**版本，一旦 MPI 运行时，系统记录了该操作，它们会立刻返回。此外，当它们返回时，消息缓冲区参数不能被用户程序访问：MPI 运行时，系统可以使用实际的用户消息缓冲区来存储消息。这样做有一个优点：消息不用从 MPI 提供的存储区中复制出来或者复制回去。

当用户程序想要重用消息缓冲区时，它可以通过调用一个 MPI 函数来强制这些操作完成。因此，非阻塞操作把一次通信分成两个阶段：

- 开始通信时调用一个非阻塞式的函数。
- 调用一个结束函数来完成通信。

每一个非阻塞式发送函数与阻塞式的版本有着相同的语法，不过非阻塞式版本在最后多了一个请求参数。例如，

```
int MPI_Isend(
      void*          msg         /* in  */,
      int            count       /* in  */,
      MPI_Datatype   datatype    /* in  */,
      int            dest        /* in  */,
      int            tag         /* in  */,
      MPI_Comm       comm        /* in  */,
      MPI_Request*   request_p   /* out */);
```

非阻塞式接收用一个请求参数替代状态参数。请求参数是实时系统中用于区分各个操作的凭据，当
347 某个程序希望结束操作的时候，结束函数就会调用请求参数。

最简单的结束函数是 MPI_Wait：

```
int MPI_Wait(
      MPI_Request*  request_p  /* in/out */
      MPI_Status*   status_p   /* out    */);
```

当它返回时，创建 * request_p 的操作也会完成。在我们的设定里，* request_p 设置为 MPI_REQUEST_NULL，* status_p 负责存储完成操作的信息。

需要注意的是，非阻塞的接收操作可用于匹配阻塞的发送操作，反之亦然。

可以使用非阻塞发送实现对最佳回路的广播。基本思想是创建一对含有 comm_sz 个元素的数组。前者用于存储新的最佳回路代价，后者用于存储请求信息，所以基本的广播操作类似于下面的代码：

```
int costs[comm_sz];
MPI_Request requests[comm_sz];

for (dest = 0; dest < comm_sz; dest++)
   if (my_rank != dest) {
      costs[dest] = new_best_tour_cost;
      MPI_Isend(&costs[dest], 1, MPI_INT, dest, NEW_COST_TAG,
            comm, &requests[dest]);
   }
requests[my_rank] = MPI_REQUEST_NULL;
```

这个循环完成后，发送操作已经开始，它们可以与基本的 MPI_Recv 匹配。

有多种方法可以处理随后的广播操作。最简单的方法可能就是调用 MPI_Waitall 函数等待之前所有的非阻塞发送。

```
int MPI_Waitall(
      int          count       /* in     */,
      MPI_Request  requests[]  /* in/out */,
      MPI_Status   statuses[]  /* out    */);
```

在此函数返回时，所有的操作应该都已经完成了（假设没有错误）。注意：在请求值为 MPI_REQUEST_NULL 时，调用 MPI_Wait 和 MPI_Waitall 都是可以的。

在 TSP 问题的静态 MPI 实现里，使用非阻塞式发送实现对最佳回路代价的广播，其性能与采用缓冲发送相比，性能如何？

6.23　MPI_Status 是一个含有成员：源（source）、标签（tag）和错误代码（error code）的结构体对象，同时它也含有消息大小的信息。但是，该参数不能被直接存取，它只能通过 MPI 的函数 MPI_Get_count 来获取： 348

```
int MPI_Get_count(
      MPI_Status*   status_p   /* in  */,
      MPI_Datatype  datatype   /* in  */,
      int*          count_p    /* out */);
```

当 MPI_Get_count 接收了消息状态和数据类型等参数后，它将返回消息中类型为 datatype 的对象的数量。因此，MPI_Iprobe 和 MPI_Get_count 可用于接收消息前计算相应消息的大小。请用这些资料编写一个 Cleanup_messages 函数，该函数在 MPI 程序退出前调用，主要用于接收未被接收的消息，使得像 MPI_Buffer_detach 这样的函数不会被挂起。

6.24　程序 mpi_tsp_dyn.c 有一个命令行参数 split_cutoff。如果某个部分回路已经访问了 split_cutoff 个或者更多的城市，那么该部分回路就不是一个可发送给其他进程的候选者。Fulfill_request 函数只发送城市数量少于 split_cutoff 的部分回路给其他进程。参数 split_cutoff 是如何影响程序的整体性能的呢？你能找到一个合理的方法来确定恰当的 split_cutoff 吗？进程数目（即 comm_sz）的改变又会怎样影响 split_cutoff 所取的最佳值呢？

6.25　MPI 程序不能发送指针（pointer），因为在发送进程中合法的地址在接收进程中可能会导致段越界的错误，或者更糟糕的是可能会访问接收进程中已经被使用的内存地址。有几种方法可以用于解决该问题：

a. 由发送者将指针所指向的对象打包到连续的存储空间，然后由接收者来解包。

b. 发送者和接收者可以创建 MPI 派生数据类型，用于映射发送者使用的存储空间和接收者的有效存储空间。

请设计两个 Send_linked_list 函数和两个与之相匹配的 Recv_linked_list 函数。第一对发送和接收函数使用 MPI_Pack 和 MPI_Unpack 函数。第二对发送和接收函数使用派生类型。第二对发送和接收函数需要发送两个消息：一个用于告诉接收者链表中结点的数量，另外一个用于发送真实的链表。比较并分析这两种方案的性能。与发送一块连续的、大小与打包后的链表一致的内存区相比，它们的性能又如何？

6.26　TSP 问题解法的动态划分 MPI 实现使用了终止检测算法，这个算法可能需要使用高精度的算术运算

349 （即有很大的分子和分母的分数计算）

a. 如果总的能量数目是 comm_sz，请解释一下为什么进程存储的能量会有格式为 $1/2^k$ 的形式，其中 $k$ 为非负整数。这种形式下，任何一个进程存储的能量除了进程 0 之外，都可以用 $k$ 来表示，$k$ 为无符号整数。

b. 请解释为什么（a）中的表达几乎不可能出现上溢或者下溢。

c. 另一方面，进程 0 需要存储一个分子不是 1 的分数。请解释如何用一个无符号整数作为分母、一个二进制数组作为分子来实现这样一个分数。这样的实现如何解决分子溢出问题？

6.27 如果 TSP 问题的动态 MPI 实现有很多进程，并且需要多次重新分配任务，那么进程 0 可能会成为能量返回的瓶颈。请解释如何使用进程的生成树，其中子进程可以发送能量给自己的父进程而不是进程 0。

6.28 修改使用了 MPI 和动态搜索树划分的 TSP 问题解决方案，以使得每个进程都会汇报各自发送“无任务可做”消息给进程 0 的次数。请推测：接收和处理“无任务可做”消息会怎样影响进程 0 总的运行时间。

6.29 在 mpi_tsp_dyn.c 中的源代码文件中含有 TSP 问题的动态分割 MPI 实现，为确定请求任务的消息应该发送给谁，在线版本使用的是 6.2.12 节中提到的三个方案的第一个方案。请用其他两种方法来解决该问题，并对比这三者各自的性能。是否有一种方法的性能总是比其他两种好？

6.30 针对 $n$ 体问题和 TSP 问题，确定 3 种 API 中哪一种更合适该问题。

a. 分析每个串行程序分别需要多少存储空间。当用并行程序来解决大问题时，它们是否可以适应共享内存系统的存储条件？对于分布式内存系统又如何呢？

b. 对于每个并行算法而言，究竟需要多少通信呢？

c. 串行程序可以很容易地使用 OpenMP 指令改写为并行程序吗？是否需要如条件变量和读写锁等同步结构？

比较你的想法和实际程序的性能。你是否做出了正确的选择？

## 6.7 编程作业

350 6.1 用典型的四阶龙格库塔（Runge Kutta）法来求解常微分方程。用这种方法代替欧拉法来估计 $s_q(t)$ 和 $s'_q(t)$。修改串行的 $n$ 体问题解法的简化版本、Pthreads 或者 OpenMP 实现的 $n$ 体问题解法，以及 MPI 实现的 $n$ 体问题解法。与使用欧拉方法所得到的输出相比，这些解法得到什么输出？比较这两种方法的性能。

6.2 修改 $n$ 体问题解法的基本 MPI 实现，使程序用一个环形传递代替对 MPI_Allgather 的调用。当一个进程收到分配给其他进程的粒子的位置信息时，它计算出它所分配到的粒子与收到信息的粒子之间由于相互作用导致的所有作用力。当收到 comm_sz − 1 组位置信息后，每一个进程应该能计算出它所管理的粒子所受到的总作用力。比较原来的基本 MPI 实现，这种方法性能如何？与简化算法的 MPI 实现相比性能又如何？

6.3 我们可以使用共享内存来模拟环形传递：

```
Compute loc_forces and tmp_forces due to my particle
    interactions;
Notify dest that tmp_forces are available;
for (phase = 1; phase < thread_count; phase++) {
   Wait for source to notify me that tmp_forces are available;
   Compute forces due to my particle interactions with
         ''received'' particles;
   Notify dest that tmp_forces are available;
}
Add my tmp_forces into my loc_forces;
```

为了实现它，主线程可以为总作用力分配 $n$ 个存储位置，为“临时”作用力分配 $n$ 个存储位置。每个线程会对两个数组中存储位置的部分子集进行操作。最容易实现“通知”和“等待”的方法是信号

量。主线程可以为每一个源 - 目标对分配一个信号量并将每一个信号量初始化为 0（或“上锁”）。在一个线程完成了作用力计算后，它可以调用 sem_post 通知目标进程；一个线程可以阻塞在调用 sem_wait 上，等待下一组作用力可用。用 Pthreads 实现这个方案。它的性能与原始的简化版本的 OpenMP/Pthreads 实现相比性能如何？与简化版本的 OpenMP/Pthreads 实现相比，这个方案的内存利用率如何？与简化的 MPI 实现相比又如何？

6.4　通过让每个进程仅存储 $n/$comm_sz 个质量，并将质量与位置和作用力信息那样传递，简化的 $n$ 体问题 MPI 实现可以进一步简化。这个实现需要为 tmp_data 数组增加附加的 $n/$comm_sz 个 **double** 型的存储空间。这个改变会如何改变解决方案的性能？与原来的 $n$ 体问题 MPI 实现相比，所需要的内存空间有什么不同？与简化版本的 OpenMP 实现相比，这个方案的内存利用率如何？ 351

6.5　树搜索的 OpenMP 动态划分实现有一个 Terminated 函数，它使用的是忙 - 等待的方式，这种方式可能会极大地消耗系统的资源。问一下系统管理员 Pthreads 和 OpenMP 是否可以在程序中一起使用。如果可以，那就修改 OpenMP 动态划分实现的代码，用 Pthreads 和条件变量来解决重新分配任务和终止问题。比较前后两种方案的性能。

6.6　我们讨论的树搜索的迭代实现使用的是基于数组的栈。修改 Pthreads 和 OpenMP 动态分割树搜索实现的程序，以使得它可以使用基于链表的栈。该方案的性能如何？

在命令行参数中加入“截止大小”，截止大小又会怎样影响系统的性能呢？

6.7　使用 Pthreads 和 OpenMP 实现一个基于共享栈的树搜索程序。正如文中所讨论的，调用 Push 和 Pop 操作全部都访问共享栈时很没有效率，所以应该在每个线程里使用一个各自的私有栈。然而，Push 函数偶尔会压入一些部分回路进入共享栈，Pop 函数也会对共享栈执行一些出栈操作（如果调用线程已经完成任务）。因此，程序中需要增加一些输入参数：

a. 把回路压入共享栈的频率。这可以是一个 **int** 型整数。例如，如果线程生成的第 10 个回路径应该压入共享栈，那么命令行参数应该是 10。

b. 入栈操作的块大小。如果入栈操作是一次压入一块（含有多个回路），而不是单一的回路进入共享栈，那么会减少很多竞争。

c. 出栈操作的块大小。如果在没有任务可做的时候，每次出栈操作都只是一个回路，那么对于共享栈而言，会有更多的竞争产生。

一个线程如何确定程序已经终止了呢？

请用 Pthreads 和 OpenMP 实现这个设计，以及你自己的终止检测算法。不同的输入参数是如何影响程序性能的？这个程序的性能与书中动态负载均衡实现的性能相比又如何呢？ 352

第7章
An Introduction to Parallel Programming

# 接下来的学习方向

既然你已经有了使用 MPI、Pthreads 和 OpenMP 来编写并行程序的基础，也许你想知道是否还有其他的方法可以使用。答案当然是有。下面是一些深入阅读的主题。对于每个主题我们都列出了一些参考文献。需要注意的是，这只是个简单列表，并行计算是一个快速变化的领域。在深入研究前，你也许需要在因特网上做一些搜索。

### 1. MPI

MPI 是一个庞大而不断发展的标准。我们只讨论了原标准中的一部分，MPI - 1。我们已经了解了一些点对点和集体通信，以及一些创建派生数据类型的辅助工具。MPI - 1 还提供了创建与管理通信子和拓扑结构的标准。我们简单地讨论了通信子；一个通信子是一组可以相互发送消息的进程。拓扑结构提供了一种手段来对通信子中的进程施加逻辑结构。例如，我们讨论了通过分配矩阵的行块给每个进程来将矩阵划分给一组 MPI 进程。在许多应用中，分配子矩阵块给进程会更加方便。在这样的应用中，有用的想法就是把所有进程想象成矩形网格，网格里的进程是由其处理的子矩阵来区分。拓扑结构为我们提供了一种区分的方法。例如，我们可以用第二行和第四列来表示进程 13。参考文献［23，43，47］提供了 MPI - 1 更详细的介绍。

与 MPI - 1 相比，MPI - 2 增加了动态进程管理、单向通信以及并行 I/O。第 2 章谈到了单向通信和并行 I/O。并且，在讨论 Pthreads 时，我们谈到许多 Pthreads 程序是在需要的时候创建线程，而不是在程序开始执行的时候就创建了所有的线程。动态进程管理为 MPI 的进程管理增加了这个功能以及其他一些功能。文献［24］是对 MPI - 2 的一个介绍。文献［37］是 MPI - 1 和 MPI - 2 标准的混合版本。

### 2. Pthreads 和信号量

353 我们已经讨论了 Pthreads 中的一些函数，例如启动和终止线程，保护临界区和线程同步。但还有一些其他函数没有讨论到，这可能会影响已讨论的函数的行为。在不同的对象初始化函数中有一个“属性”（attribute）参数，一般情况下都是简单地传递 NULL 值给这样的参数，所以对象函数会采取“默认”行为。如果传递其他值给这些参数，那么这些函数会表现出不同的作用。例如，如果一个线程含有一个互斥量，并且想对互斥量再次执行锁操作（例如递归函数调用），默认情况下该情况是未定义的。然而，如果互斥量是用属性 PTHREAD_MUTEX_ERRORCHECK 来创建的，那么第二次对 pthread_mutex_lock 的调用就会返回错误。另一方面，如果互斥量是用属性 PTHREAD_MUTEX_RECURSIVE 来创建的，那么该操作就会成功返回。

另外，我们也接触到了非阻塞式操作的使用。在讨论树搜索动态实现的时候，我们提及了 pthread_mutex_trylock 函数。同样地，也有非阻塞版本的读写锁函数和 sem_wait。这些函数为线程提供了当锁或者信号量被其他线程占用的情况下可以继续工作的机会。因此它们有极大的潜力去提升应用程序的并行度。

与 MPI 一样，Pthreads 和信号量也是在不断发展的标准。它们是 POSIX 标准的一部分。网络上有 POSIX 标准的最新版本，可以通过 Open Group［41］来寻找。Pthreads 头文件的帮助手册［46］和信号量头文件的帮助手册［48］提供了 Pthreads 和信号量所有函数的帮助手册链接。关于 Pthreads 的某些文档，参见文献［6，32］。

### 3. OpenMP

我们已经了解了一些 OpenMP 中最重要的指令、子句和函数，也知道了如何去开启多个线程，如何并行化 `for` 循环，如何保护临界区资源，如何调度循环，以及如何修改变量的作用域。然而，依然有一些很有用的知识我们没有涉及。其中最重要的一个新指令是最近被引入的 `task` 指令。它可以用于并行化如递归函数调用和 `while` 循环这样的结构。本质上，它将一个结构化块标记为一个线程执行的任务。当一个线程遇到 `task` 指令时，线程可以或者执行结构化块中的代码，或者被加入到一个任务概念池中。当前组中的线程将执行任务，直到任务概念池变空。

OpenMP 体系结构审查委员会正在不断开发 OpenMP 标准。最新的文档可从文献［42］中获得。文献［8，10，47］也是一些相关的资料。

### 4. 并行硬件

并行硬件发展很快。幸运的是，来自于 Hennessy 和 Patterson［26，44］的文档也在以同样的速度更新。他们提供了诸如指令级并行、共享内存系统和网络互连等方面主题的全面概述。文献［11］专门关注和研究了并行系统。

### 5. 通用并行编程

有许多介绍并行编程的书籍并不只关注某类特定的 API。文献［50］提供了一个相对初级 354
的、既关于分布式内存系统也关于共享式内存系统编程的讨论。文献［22］对并行算法进行了广泛的讨论，并对程序性能给出了先验分析。文献［33］概述了目前流行的并行编程语言，并给出了将来的一些发展方向和研究热点。

对于共享式内存系统的编程可以参考文献［3，4，27］。文献［4］讨论了设计和开发共享式内存系统程序的技术。除此之外，它也开发了一些为解决如搜索和排序问题的并行程序。文献［3］和文献［27］都深入讨论了如何确定一个算法是否正确以及确保正确的机制。

### 6. GPU

在并行计算中最有前途的发展方向之一就是使用 GPU 进行通用计算。它已经很成功地应用于不同的数据并行程序，或者处理器之间通过划分数据来获取并行度的程序。文章［17］和书［30］概述了 GPU 的相关知识。除此之外，书［30］还提供了关于 CUDA 编程的介绍、对 GPU 进行编程的 NVIDIA 语言，以及 OpenCL（一种在包括常规 CPU 和 GPU 的异构系统上进行编程的标准）。

### 7. 并行计算的历史

很多程序员会惊讶地发现，其实并行计算与大部分的计算机科学的主题一样，有着漫长而古老的历史。文章［14］给出了一些简单的调研和引用。而网站［51］在并行系统的发展历史上是有里程碑意义的。

好吧，现在你可以开始征服新的世界了！ 355

# 参考文献


[1] S.V. Adve, K. Gharachorloo, Shared memory consistency models: A tutorial Digital Western Research Laboratory research report 95/7. <ftp://gate-keeper.dec.com/pub/DEC/WRL/research-reports/WRL-TR-95.7.pdf>, 1995 (accessed 21.09.10).

[2] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the American Federation of Information Processing Societies Conference, vol. 30, issue 2, Atlantic City, NJ, 1967, pp. 483–485.

[3] G.R. Andrews, Multithreaded, Parallel, and Distributed Programming, Addison-Wesley, Boston, 2000.

[4] C. Breshears, The Art of Concurrency: A Thread Monkeys Guide to Writing Parallel Applications, O'Reilly, Sebastopol, CA, 2009.

[5] R.E. Bryant, D.R. O'Hallaron, Computer Systems: A Programmer's Perspective, Prentice Hall, Englewood Cliffs, NJ, 2002.

[6] D.R. Butenhof, Programming with Posix Threads, Addison-Wesley, Reading, MA, 1997.

[7] B.L. Chamberlain, D. Callahan, H.P. Zima, Parallel programmability and the Chapel language, Int. J. High Perform. Comput. Appl. 21 (3) (2007) 291–312. SAGE Publications, Thousand Oaks, CA, see also <http://chapel.cray.com/> (accessed 21.09.10).

[8] R. Chandra, et al., Parallel Programming in OpenMP, Morgan Kaufmann, San Francisco, 2001.

[9] P. Charles, et al., X10: An object-oriented approach to non-uniform clustered computing, in: R.E. Johnson, R.P. Gabriel (Eds.), Proceedings of the 20th Annual ACM Special Interest Group on Programming Languages (SIGPLAN) Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), San Diego, 16–20 October, ACM, New York, 2005, pp. 519–538. See also <http://x10-lang.org/> (accessed 21.09.10).

[10] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, MIT Press, Cambridge, MA, 2008.

[11] D. Culler, J.P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, San Francisco, 1998.

[12] J. Demmel, Hierarchical methods for the $N$-body problem, Lecture 22 for Demmel's spring 2009 class, "Applications of Parallel Computing," CS267/EngC233, UC Berkeley. <http://www.cs.berkeley.edu/~demmel/cs267_Spr09/Lectures/lecture22_NBody_jwd09.ppt>, 2009 (accessed 21.09.10).

[13] E.W. Dijkstra, Cooperating sequential processes, in: F. Genuys (Ed.), Programming Languages, Academic Press, New York, 1968, pp. 43–112. A draft version of this paper is available from <http://userweb.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF> (accessed 21.09.10).

[14] P.J. Denning, J.B. Dennis, The resurgence of parallelism, Commun. ACM 53 (6) (2010) 30–32.

[15] E.N. Dorband, M. Hemsendorf, D. Merritt, Systolic and hyper-systolic algorithms for the gravitational N-body problem, with an application to Brownian motion, J. Comput. Phys. 185 (2003) 484–511.

[16] Eclipse Parallel Tools Platform. <http://www.eclipse.org/ptp/> (accessed 21.09.10).

[17] K. Fatahalian, M. Houston, A closer look at GPUs, Commun. ACM 51 (10) (2008) 50–57.

[18] M. Flynn, Very high-speed computing systems, Proc. IEEE 54 (1966) 1901–1909.

[19] I. Foster, Designing and Building Parallel Programs, Addison-Wesley, Reading, MA, 1995. Also available from <http://www.mcs.anl.gov/~itf/dbpp/> (accessed 21.09.10).

[20] I. Foster, C. Kesselman, (Eds.), The Grid 2, second ed., Blueprint for a New Computing Infrastructure, Morgan Kaufmann, San Francisco, 2003.

[21] The GNU MP Bignum Library. <http://gmplib.org/> (accessed 21.09.10)

[22] A. Grama, et al., Introduction to Parallel Computing, second ed., Addison-Wesley, Harlow, Essex, 2003.
[23] W. Gropp, E. Lusk, A. Skjellum, Using MPI, second ed., MIT Press, Cambridge, MA, 1999.
[24] W. Gropp, E. Lusk, R. Thakur, Using MPI-2, MIT Press, Cambridge, MA, 1999.
[25] J.L. Gustafson, Reevaluating Amdahl's law, Commun. ACM 31 (5) (1988) 532–533.
[26] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, fourth ed., Morgan Kaufmann, San Francisco, 2006.
[27] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, Boston, 2008.
[28] IBM, IBM InfoSphere streams v1.2.0 supports highly complex heterogeneous data analysis, IBM United States Software Announcement 210-037, February 23, 2010. Available from <http://www.ibm.com/common/ssi/rep_ca/7/897/ENUS210-037/ENUS210-037.PDF> (accessed 21.09.10).
[29] B.W. Kernighan, D.M. Ritchie, The C Programming Language, second ed., Prentice-Hall, Upper Saddle River, NJ, 1988.
[30] D.B. Kirk, W-m.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, Boston, 2010.
[31] J. Larus, C. Kozyrakis, Transactional memory, Commun. ACM 51 (7) (2008) 80–88.
[32] B. Lewis, D.J. Berg, Multithreaded Programming with Pthreads, Prentice-Hall, Upper Saddle River, NJ, 1998.
[33] C. Lin, L. Snyder, Principles of Parallel Programming, Addison-Wesley, Boston, 2009.
[34] J. Makino, An efficient parallel algorithm for $O(N^2)$ direct summation method and its variations on distributed-memory parallel machines. New Astron. 7 (2002) 373–384.
[35] J.M. May, Parallel I/O for High Performance Computing, Morgan Kaufmann, San Francisco, 2000.
[36] Message-Passing Interface Forum. <http://www.mpi-forum.org> (accessed 21.09.10).
[37] Message-Passing Interface Forum, MPI: A message-passing interface standard, version 2.2. Available from <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf> (accessed 21.09.10).
[38] Microsoft Visual Studio 2010. <http://msdn.microsoft.com/en-us/vstudio/default.aspx> (accessed 21.09.10).
[39] Message-Passing Interface Forum, MPI: A message-passing interface standard. Available from <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html> (accessed 21.09.10).
[40] L. Oliker, et al., Effects of ordering strategies and programming paradigms on sparse matrix computations, SIAM Rev. 44 (3) (2002) 373–393.
[41] The Open Group. <http://www.opengroup.org> (accessed 21.09.10).
[42] OpenMP Architecture Review Board, OpenMP application program interface, version 3.0, May 2008. See also <http://openmp.org/wp/> (accessed 21.09.10).
[43] P. Pacheco, Parallel Programming with MPI, Morgan Kaufmann, San Francisco, 1997.
[44] D.A. Patterson, J.L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, fourth ed., Morgan Kaufmann, Boston, 2009.
[45] Project Fortress Community. Available from <http://projectfortress.sun.com/Projects/Community> (accessed 21.09.10).
[46] `pthread.h` manual page. Available from <http://www.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html> (accessed 21.09.10).
[47] M. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill Higher Education, Boston, 2004.
[48] `semaphore.h` manual page. Available from <http://www.opengroup.org/onlinepubs/000095399/basedefs/semaphore.h.html> (accessed 21.09.10).
[49] Valgrind home. <http://valgrind.org> (accessed 21.09.10).
[50] B. Wilkinson, M. Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, second ed., Prentice-Hall, Upper Saddle River, NJ, 2004.
[51] G. Wilson, The history of the development of parallel computing. Available from <http://parallel.ru/history/wilson_history.html>, 1994 (accessed 21.09.10).

# 索　引[⊖]

索引中的页码为英文原书页码，与书中页边标注的页码一致



⊖ 页码后面的 "*f*" 表示图, "*t*" 表示表, "*n*" 表示脚注。